TECHNISCHE UNIVERSITÄT
CHEMNITZ

Automatic Kernel Intercommunication Interface for a Simulation Platform

Master Thesis

Submitted in fulfillment of the requirements for the academic degree
M.Sc. in Automotive Software Engineering

Professorship of Computer Engineering
Faculty of Computer Science

Submitted by: Umair Latif
Matrikel-Nr.: 307431
Supervisors: Prof. Dr. rer. nat. Wolfram Hardt
Dr. Ing. Alexander Zaytsev
(LuK GmbH & Co. KG)

Bühl, January 2017

Abstract

CluSys is a simulation platform that is developed by LuK GmbH & Co. KG to support product development engineers. It integrates different simulation calculation kernels and provides a collaborative environment for tool-independent chain simulation processes. The data between simulation modules in chain simulations is exchanged by defining intercommunications between individual parameters manually. This thesis provides a concept to automate the data exchange process between the parameters of different simulation models based on their physical meaning. Different approaches implemented in contemporary simulation environments are discussed, analyzed and a customized solution compatible with CluSys based on *abstract interface* approach is developed. This concept is implemented using an *SQL* database creating a *parameter dictionary* and an automatic intercommunication interface that uses the dictionary to recognize the parameters and define data exchange processes between them automatically. The implemented solution makes the data exchange between parameters of different simulation models independent of their labels, module names or units.

Acknowledgements

First, I would like to thank my supervisor Dr. Ing. Alexander Zaytsev at LuK GmbH & Co. KG, for his support throughout the whole project. His helpful and dedicated attitude and his insightful remarks allowed me to complete this master thesis successfully. From the very first day, he always listened to my questions patiently and helped me out, whether if it was about the coffee machine or the complex class structure of CluSys. I learned a lot from him in these six months.

I would also like to thank my department manager, Mr. Bertrand Pennec, my team lead Dr. Mathias Müller and all the team members of EBP1, especially Frederic, Ling and Marc, who supported me throughout my project. Their valuable suggestions and friendly attitude during team meetings and presentations always motivated me.

On the academic side, I would like to thank Prof. Dr. rer. nat. Wolfarm Hardt, who always has been a source of great inspiration for me. I thank him for his teachings, valuable comments and suggestions, especially on my concept presentation. Furthermore, I would like to thank Mr. Frank Ullmann, who coordinated with me throughout my master thesis. His detailed remarks on the presentations and documentation were beyond helpful.

At this point, I would like to thank my family and friends, who supported me in person or with their moral support throughout my work. Last but not least, I would like to pay my special regards to my parents, who are miles away but their love and encouragement always keep me going. I couldn’t thank enough for their love.

Table of Contents

List of Figures

List of Tables

Abbreviations

Full Form	Abbreviation
Original Equipment Manufacturer	OEM
Kinematics of Engagement System	KES
Load Adjustment Clutch	LAC
Torque Fluctuation Analysis	TUCAN
Computer Aided Engineering	CAE
Functional Mockup Unit	FMU
Netherlands Space Office	NSO
European Space Agency	ESA
Graphical User Interface	GUI
Database Management System	DBMS
Structured Query Language	SQL
Non-SQL or Not only SQL	NoSQL
Common Language Infrastructure	CLI
Microsoft Intermediate Language	MSIL
Unified Modelling Language	UML
Extensible Markup Language	XML
Common Language Runtime	CLR or clr
English: Double Clutch	DK

Table 1: Table of abbreviations

Chapter 1

1. Introduction

The automotive industry has grown rapidly during the last few decades and ever-changing trends in mobility require a robust and efficient product development process. Along with this, an everlasting desire for shorter development time with quality improvement is still there. Therefore, the concept of "virtual product development" using simulation techniques is taking the place of lengthy process of expensive experimentation with real prototypes. With the advancement of simulation technology in the field of multibody simulations, it is used in wide variety of fields, automobile industry being the major player. Simulation technology gives a huge edge to the companies because it does not require frequent prototyping and time to market could be reduced efficiently, not to mention the optimized product, which is the core benefit. With ever more powerful computers, it is possible to create accurate dynamic models using the already present mathematical and physical foundations [1]. Due to this factor, research in simulation technologies and simulation software is the major field for the past few decades. The aim of an efficient simulation software is not only to calculate and simulate the behavior of a physical system but also to complement the overall abilities of a product development engineer by achieving a higher productivity level due to simpler, interactive and robust user interface.
The aim behind this thesis is to develop further such a simulation platform CluSys used in industry. The thesis involves research, concept and development of an automatic intercommunication interface for the simulation platform CluSys. In this chapter, first the motivation behind this thesis work will be stated which will give an overview about the current situation and the problem. The objectives of this thesis work will be given and in the last section, the breakdown of this thesis document will be described.

1.1 Motivation

The company LuK GmbH und Co. KG is part of the Schaeffler Group, which is a global integrated automotive and industrial supplier. For the past five decades, LuK has been providing systems and components for the automotive drive train, for example clutch systems, dampers and transmission components, etc. to customers around the globe. Today, approximately every third automobile has a clutch system produced by LuK. The product development team in LuK designs and optimizes the products to the last details to match customer requirements and high industry standards.

For this optimized designing and product development process, LuK engineers use different state-of-the-art commercial simulation software, for example MATLAB ©, SimulationX ©, etc. With these simulation tools, clutch systems and other LuK products are modelled using complex mathematical algorithms. These simulation models are called calculation kernels. To avoid redundant development of these calculation kernels in different departments and to standardize the product development process throughout the company, a simulation platform CluSys (Clutch Systems) has been developed by Simulation Development department. The platform also promotes a modular approach to product development cycle. These calculation kernels are imported in the platform and the users can create, access and use different simulation modules based on them. CluSys work as a unified control program for these calculation kernels. It takes user input, calculates the results using these calculation kernels and provides simulation results to the user. In chain simulations, these calculation kernels need to intercommunicate among themselves to exchange results from one module to another during the simulation process. This intercommunication between modules is defined manually in CluSys.

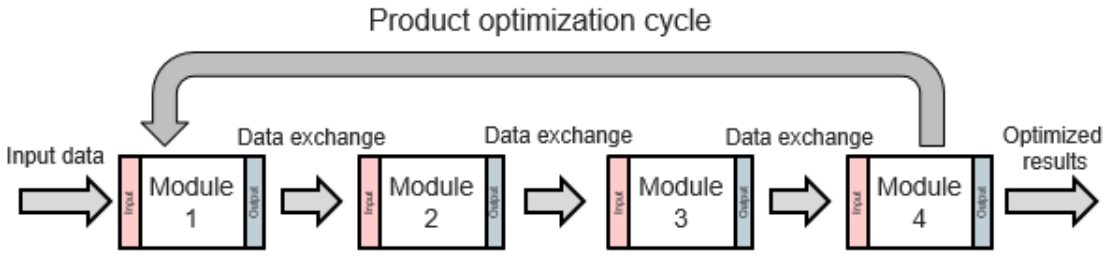


Figure 1: A chain simulation process

For complex simulation problems, more than one simulation modules are used to produce specific results. In CluSys there is a possibility of creating a simulation toolchain where user input or product specific data from a database is fed to one simulation module and the manipulated output from this module and the user entered data are then fed to the next module, then to the next one and so on to get the desired output (see Figure 1). An intercommunication between these calculation kernels for the exchange of data is created in order to transfer the values of individual parameters from one module to another.

The manual intercommunication between calculation kernels takes a lot of time and effort. It makes the chain simulation process inefficient and less productive. The simulation modules could have hundreds of parameters and the user has to search through these parameters in both modules for each interconnection. Due to the current modular

development approach, different simulation modules are developed by experts in different departments of the company and sometimes a specific module of the system could even be provided by an external company. This requires a deep knowledge of simulation modules on the part of users, which makes the simulation platform unattractive for the users other than simulation experts.

1.2 Objectives of the Thesis

The main objective of this master thesis is to automate the above-mentioned intercommunication between different calculation kernels during a chain simulation process. This automatic intercommunication process should be abstract and data exchange between calculation kernels should be based on underlying physical meaning of the parameters of the calculation kernels in a chain simulation project regardless of their name, module name or physical unit.

In order to develop such a mechanism for recognizing the physical meaning behind a parameter, state of the art in commercially available simulation platforms should be analyzed and a comparable solution for CluSys should be realized.

The automatic process functionality should be placed parallel to the existing manual data processing mechanism and therefore should be fully integrated in the CluSys platform. The already developed architecture of the platform should be kept in mind.

1.3 Overview of the Document

This document is divided into following further chapters:

Chapter 2 provides background knowledge by introducing a calculation kernel, the simulation platform CluSys and explaining a real-life example of a simulation toolchain in CluSys. This would help in understanding the problem more deeply and help in comprehending the requirements and the concept of the proposed solution.

Chapter 3 covers the state of the art in simulation development field. A comparison of commercially and off-the-shelf available simulation modelling software and platforms is given. Current solution for intercommunication between simulation modules in CluSys is described in detail along with its drawbacks.

In chapter 4, the researched methods and techniques in contemporary simulation environments are analyzed. In the end requirements for the completion of thesis are given.

In chapter 5, the programming environment and technologies are discussed briefly. The simulation platform CluSys with respect to its structure is discussed and communication

between its modules is explained. Different database technologies are discussed and a brief comparison with respect to our requirements is done as well.

In chapter 6, the concept for the solution based on the analysis of the state of the art is proposed. The concept is divided into three main parts and is discussed in detail.

Chapter 7 covers the implementation phase of the solution. The implementation is based on the concept discussed in the sixth chapter.

Chapter 8 describes the testing phase with examples of some tests and their results with respect to the implementation.

In chapter 9, the complete project is concluded with a summary of the whole work. Finally, the prospects for further development are suggested in the outlook section.

Chapter 2

2. Background

In this chapter the basic knowledge about the used terms, concepts and the software platform will be given. This knowledge will help the reader to understand the remaining document. First, the software platform CluSys will be introduced, then a calculation kernel will be explained with an example and in the end a detailed example of a simulation toolchain or a chain simulation process will be explained which will help to understand the state of the art and the problem.

2.1 Simulation Platform - CluSys (Clutch Systems)

CluSys (Clutch Systems) is an in-house project of the company and is being continuously developed and maintained by the Development Simulation *(German: Entwicklung Berechnung)* department since 2000. CluSys is used by over 400 users in different company branches worldwide.

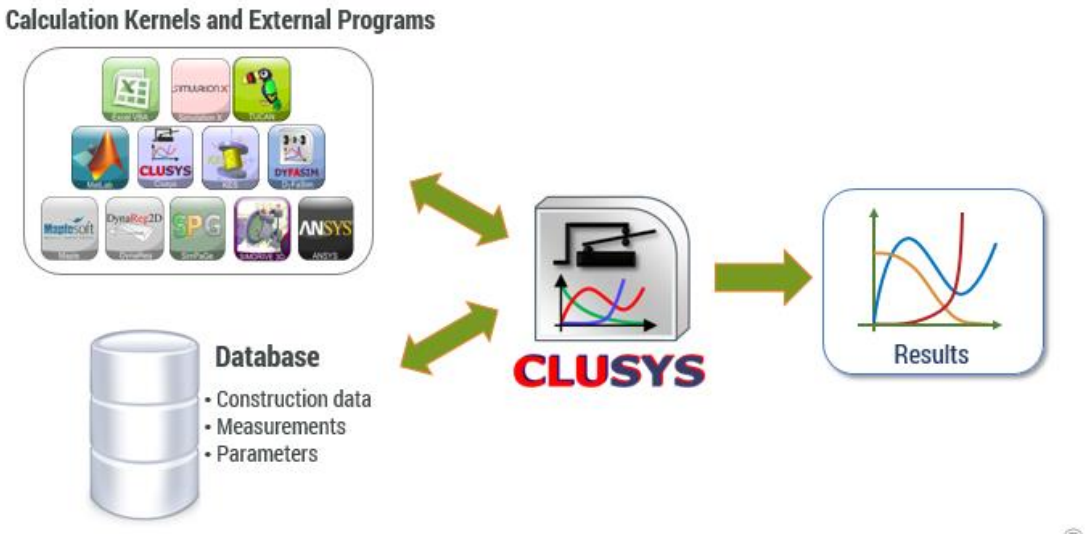


Figure 2: Simulation platform CluSys

It facilitates company-wide standard product development process with the help of tools like databases and user/group profiles (see Figure 2). It is used by LuK engineers to design and optimize individual clutch parts as well as total clutch systems according to the specified customer requirements. CluSys enables the calculation based on already developed simulation modules using the product specific data from users. The calculations based on different mathematical models are done using external solvers, either commercial

or as runtime libraries integrated into CluSys as *calculation kernels*. CluSys serves as the control program for these calculation kernels and takes care of the input, running the simulation based on the user input and representation of the simulation results with respect to their formats. CluSys provides many benefits to the user, some of them are as following: [2]

- CluSys user does not need knowledge of simulation software, such as MATLAB, SimulationX or DyFaSim[1] to work with simulation modules. These modules are already integrated in the CluSys and could be controlled from within the software with simpler and intuitive user interfaces
- LuK engineers do not need license to use these modules, as would have been the case if the simulation modules were controlled and used from the respective commercial software
- Simulation results could be visualized within the CluSys program, as well as with external software
- All standardized simulation modules would be available for reuse in all company branches and departments, increasing productivity and standardized product development

CluSys has two different user modes, *Expert* and *End-user*.

2.1.1 Expert mode

Expert mode is used by the simulation experts in Simulation Development department to create simulation packages for end-users. In Expert mode, experts can define parameters and connect them to a certain input or output from a calculation kernel or simulation module. They can define different properties for these parameters to facilitate the end user. Experts can create graphical user interfaces for each individual part of a complete system. Once a module is completely integrated in the CluSys platform, it is rolled out to end users.

2.1.2 End-User mode

End users are the engineers in Product Development department. For end users the simulation module is abstract like a *container* and they can only see the input

[1] DyFaSim (*German: Dynamische Fahrzeug Simulation*) is in-house multibody modelling tool of LuK GmbH & Co. KG

parameters and observe the results generated by that *container*. End users can change the pre-defined parameter values to change the behavior of the simulation and use these modules in various projects with different combinations and specifications to analyse different results. Figure 3 shows a screenshot of the CluSys environment with a simulation module.

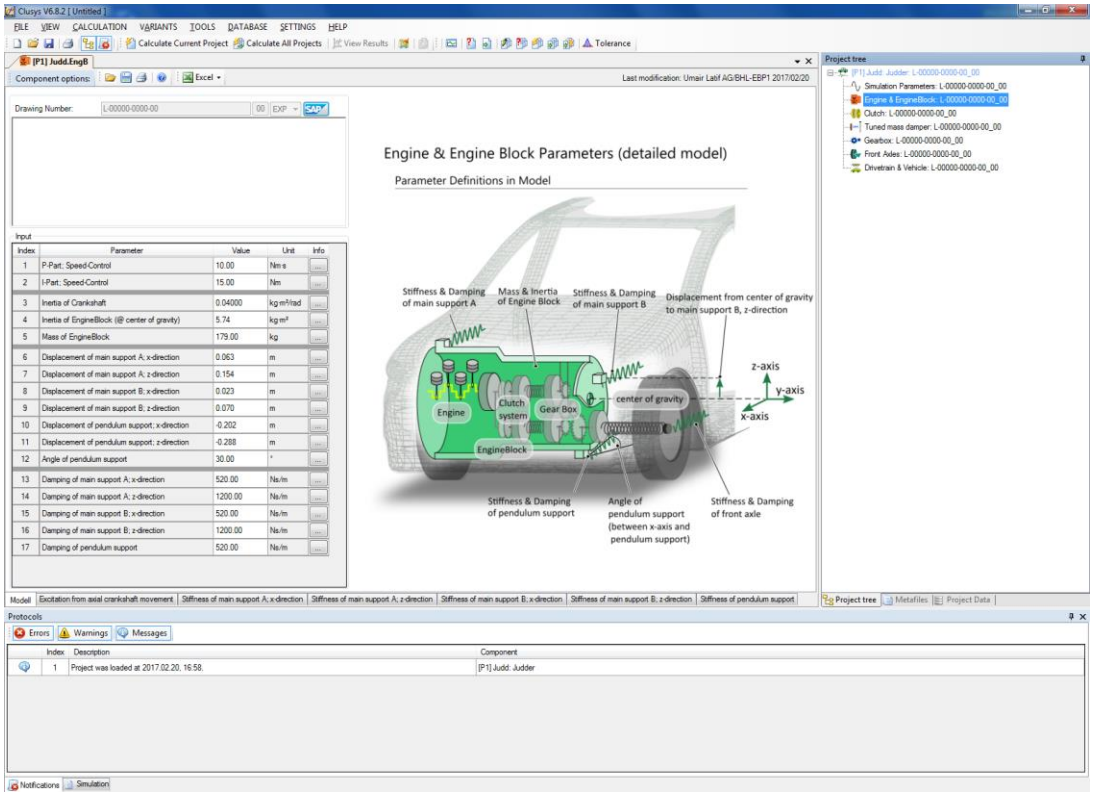


Figure 3: CluSys environment

2.2 What is a Calculation Kernel?

First, a definition of simulation process is quoted in the following excerpt:

> *"Simulation is the process of designing a model of a real or imagined system and conducting experiments with that model. The purpose of simulation experiments is to understand the behaviour of the system or evaluate strategies for the operation of the system. Assumptions are made about this system and mathematical algorithms and relationships are derived to describe these assumptions - this constitutes a "model" that can reveal how the system works."* [3]

By this definition, a simulation consists of two parts, a *simulation model* and a mechanism or *kernel* that '*runs'* the model. In the case of mechanical systems, such as clutch systems, models are developed based on complex physical principles, mathematical equations and

graphical representations of the systems. These models are mostly created using the *multibody*[2] simulation approach, in which each real object is composed of rigid or elastic *bodies*, which interact with each other [4]. In CluSys, a calculation kernel creates an encapsulation over these complex simulation models and the user is given an opportunity to use them by defining critical parameters for the models. For example, these calculation kernels have:

- input parameters which normally represent a specific measurement of an object (a physical quantity), a physical characteristic of the material or some specific value for the simulation
- output parameters which represent the results of the simulation, characteristic of a specific component or the whole system in the form of characteristic curves

These parameters are changed to simulate different variants according to the customer requirements. These simulation modules are developed by company's simulation engineers who are expert in their respective domains and these models are usually designed to be run independently to simulate a particular phenomenon based on the inputs from the user.

Example:

To simulate a mechanical system such as an automobile clutch system, complex mathematical and physical models are used to simulate different parts of the system. The models of mechanical parts are combined to create a complete simulation module. For example, different parts of a clutch are modelled separately and then combined to create a complete clutch model. These models have different parameters that represent the physical measurements, characteristic curves and constants values. In Figure 4, a double clutch developed by company LuK, named Load Adjustment Clutch (LAC) [5] is shown. On the left side is an actual picture of a clutch assembly. On the right side is a graphical representation of a cross section of the same clutch, showing different parts in their relative dimensions and position.

[2] In mechanics the term *multibody* refers to a system of simplest physical *bodies* or *objects* interacting together [1]

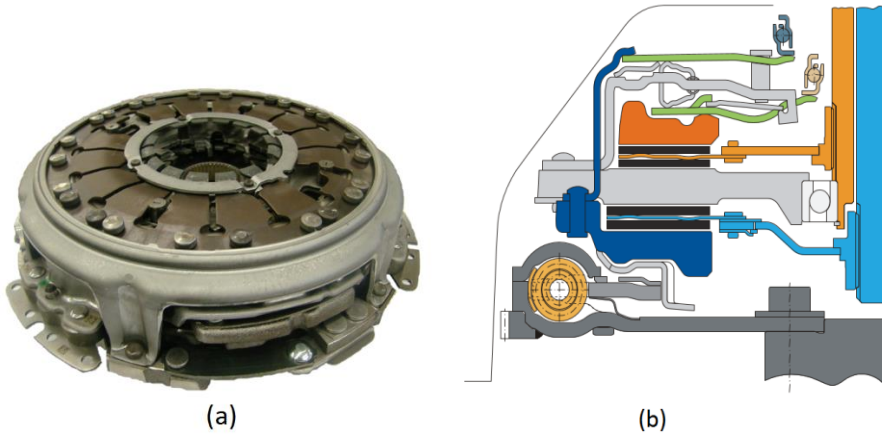


Figure 4: (a) A Load Adjustment double clutch. (b) Graphical representation of a cross section with different parts [5]

These different parts of a clutch have certain parameters, which should be given valid values to simulate the behavior of the analyzed object. For example, a clutch disc has an inner and outer diameter, which should be entered to simulate the behavior with a specific disc diameter. In the following figure (Figure 5), a dialog can be seen in CluSys in which a representation of the LAC disc of the above-described system is given along with a list of parameters on the left side.

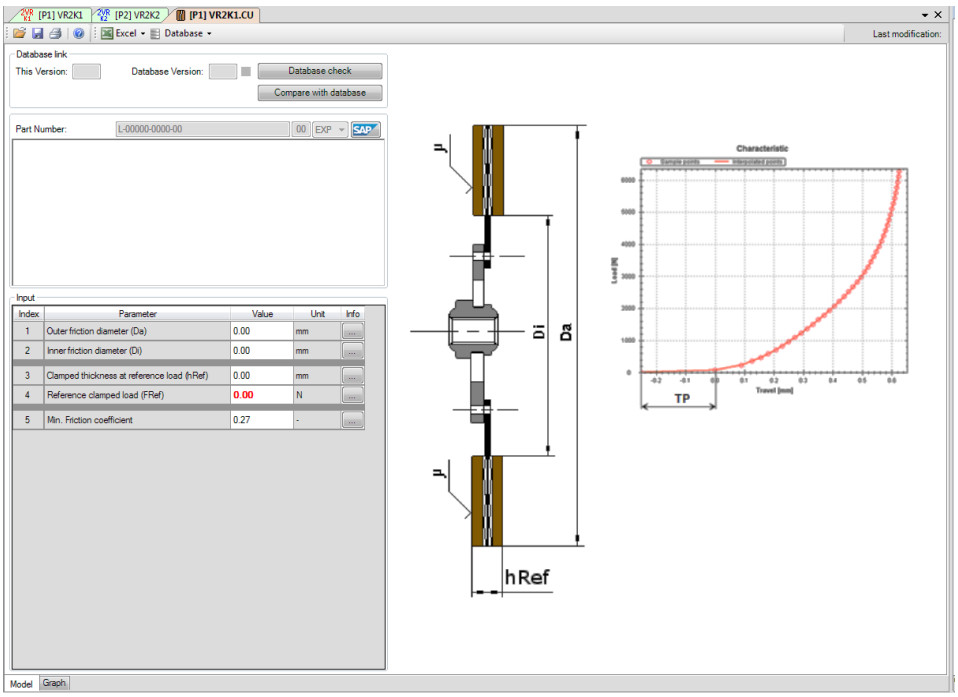


Figure 5: A representation of a clutch disc with list of parameters in CluSys [5]

2.3 An Example of Chain Simulation in CluSys

Through simulation, a record about the working of a system could be created and observed to infer decisions based on that data [4]. Nowadays, simulation is the most used methodology for problem solving in diverse range of fields, from management and statistics to simulate complex engineering problems. There are different types of simulation methodologies that are used to simulate problems and their solutions by different commercial tools, depending on their field. The two most popular methods are *continuous* and *discrete-event* simulation. In continuous simulation process, the change in the system state is recorded continuously with time, which means there is infinite number of system states, whereas, in discrete-event simulation model, the state of the system is measured at discrete events in time, only when an event occurs [4]. The calculation kernels in CluSys are based on both methodologies and the chain simulation process in CluSys is independent of the type of simulation. The results from one module are calculated and then they are fed to the next module.

In the following section, a real life example of a simulation problem from automotive industry is described. First, the problem statement would be stated and then different blocks of the simulation toolchain are explained briefly.

2.3.1 Judder

In an automobile, when a clutch is *engaged*, the torque from the engine is transferred to the drive train. This is called a "*slipping phase"*. If the drive train was stationary at the time, it would be accelerated slowly to match the engine speed. During this phase, the torque of the drive train changes continuously and drive train experiences noticeable vibrations (Figure 6). These vibrations are transferred to the wheels of the automobile and thus to the vehicle body. These vibrations are known as judder or in some literature also as *chatter* [6].

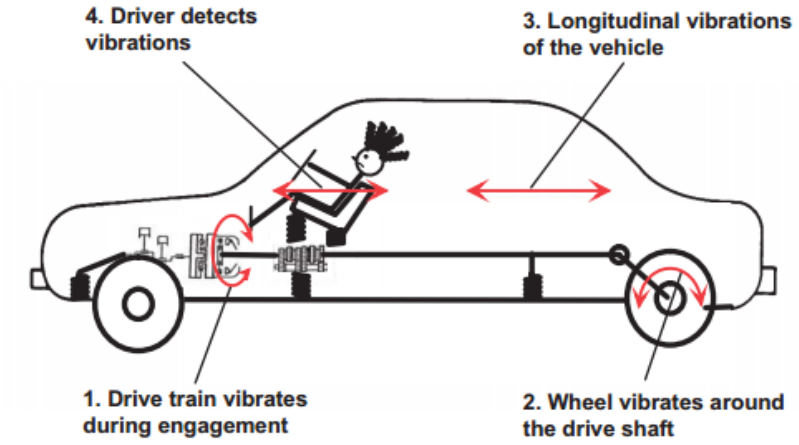


Figure 6: Judder [6]

This is a serious problem in automobile development and various methods and designs are applied to understand and overcome the judder. The vibrations caused by judder not only depend on the clutch and the clutch engagement system in use but also on the physical properties of the engine, the drive train and the overall assembly of the automobile transmission as well. That is why, it is important to study the judder problem in relation to the entire powertrain system [6].

During development in CluSys, simulation experts use different simulation modules together, creating a simulation toolchain, to evaluate judder for a specific clutch and clutch engagement system. They change certain parameters in the modules or replace the whole module with a different version to see the influence on whole system.

2.3.2 Simulation toolchain for judder

In CluSys, a typical example of a simulation toolchain to simulate judder for a specific clutch type consists of four modules. These modules simulate different systems and their behavior according to the inputs.

I. Clutch system

This module simulates a specific clutch type with customer defined measurements and values. In general, there are a number of clutch modules developed by the simulation engineers to simulate specific products. These are used in combination with other modules to study effects on judder. These modules have certain parameters, which define measurements of clutch assembly with respect to a certain automobile, for

example clutch stiffness, clutch size, etc. A typical example of such a clutch module, *Load Adjustment Clutch*, is given in first chapter.

Input: clutch geometry and stiffness properties of individual clutch parts

Output: clutch characteristic curves, such as engagement force required to open or close the clutch, torque etc.

II. KES - kinematic behavior of a clutch engagement system

This module is designed to estimate the kinematic behavior of clutch engagement system in use. Different types of engagement systems are used with respect to customer requirements, which compensate the torque modulation during clutch engagement [7] [3]. This module takes the values of the clutch module and simulates the effect of an engagement system on the clutch with respect to change in torque.

Input: system configuration, engagement system specifications, clutch type and clutch parameters, errors, etc.

Output: characteristic curves describing effects on the system due to variations, such as torque modulation with respect to mean torque

III. TUCAN - torque fluctuation analyses

TUCAN is used to study the torque fluctuations in the clutch systems. This module simulates the torque fluctuation based on the specifications of the clutch system and the deviations resulting from the pressure on the engagement system. The results (see Figure 7) from this module are used to analyze and compare the fluctuations for different clutch characteristics [8].

Input: Clutch parameters, deviations in clutch parts

Output: characteristic curves describing torque fluctuations with respect to the deviations

[3] These simulation modules are in-house developments and are not commercially available. The reference literature for these modules is only accessible via company's Intranet.

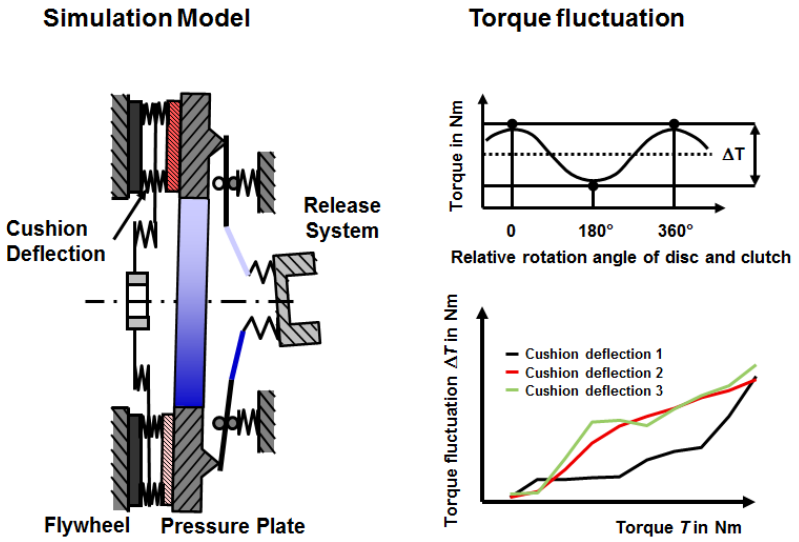


Figure 7: TUCAN simulation module and torque fluctuation characteristic curves [8]

IV. Drive train dynamics

This module is based on the actual measurements and characteristics of the automobile, for which the simulation analysis is being done. To estimate the torque delivered from engine to the drive train and the resulting judder, certain aspects of the actual vehicle are very important. This module takes care of these aspects, for example weight and dimensions of the vehicle, tires, characteristics of the drive shaft, air resistance, etc.

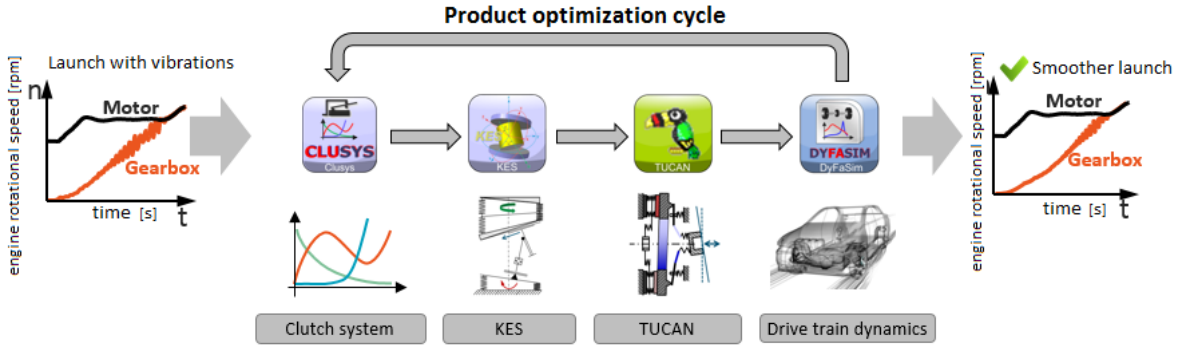


Figure 8: Simulation toolchain for Judder

In the above figure (Figure 8), a simulation toolchain for judder can be seen. In this simulation toolchain, the clutch system module gets the input data for different characteristics of a clutch and this data is processed to simulate the behavior of a

clutch. The resulting data is then fed to the kinematic engagement system module, so that it can simulate the behavior of the engagement system on the specific clutch system data. The results from this module would contain the effects of an engagement system on the torque transferred by the clutch [8].

These results would then be transferred to TUCAN module to calculate the torque fluctuation based on different factors. These fluctuations would then be transferred to the vehicle specific data from drive train specifications to see the actual effect of fluctuations on the judder. Such simulations are run multiple times to generate and compare data from different product specifications.

All the above-mentioned data transfer between clutch system, KES module and the TUCAN module is done manually by the experts. Before running the simulation, they have to define transfer of data for each parameter.

2.3.3 Module replacement

Judder is a complex phenomenon and a lot of research is done in the automobile industry to overcome it. With time, more value is being given to the comfort of the driver and the passengers in the automobile industry and nowadays even a slight amount of judder could be detected and causes discomfort for the driver. The above example shows only one type of clutch system. For better analyses of the judder problem with respect to different clutch systems and their combination with a certain engagement system, the experts have to replace these modules frequently to record the effects of each specification. These results are then compared to choose the best design for a current automobile type.

If an expert replaces a clutch module from this simulation toolchain with a different one, the intercommunication will be lost between the clutch and the engagement system (KES) module and the data transfer will not be possible. The data transfer has to be defined again manually to run the simulation again. Such type of complex problem analyses requires sufficient amount of data and this creates a huge problem if the data transfer has to be defined again for each parameter, every time a module is changed.

Summary

The software platform CluSys provides a collaborative environment for simulation of multiple simulation models together. These simulation models are developed in different departments of

the company using specialized simulation software. Each simulation model has its own goals and functions. These models are packaged together as *simulation modules* or *calculation kernels* and then integrated in CluSys for end users (for example, product development engineers). Simulation experts create these simulation packages as per user requirements and release them for users. For complex automotive problems, such as *judder*, a chain simulation process is done and it needed to be repeated with different module combinations to achieve a certain goal in terms of simulation results. Such chain simulation processes require frequent module replacements within a project, which due to the manual interconnection between these calculation kernels, is tedious and requires a lot of data processing before a simulation package could be released for the end users.

Chapter 3

3. State of the Art

In this chapter, a brief discussion about commercially available simulation software with examples will be given. The state of multibody simulations toolchain will be discussed with already used approaches for intercommunication and parameter exchange between simulation modules. These approaches will be analyzed to suggest the solution with respect to the current state of in-house simulation platform CluSys.

3.1 Multibody Simulation Software

Commercial multibody simulation software now offers a great range of tools for not only the rigid body simulations, but for elastic bodies as well [9]. As could be seen in the example given in previous chapter, problems in automotive industry usually need combination of simulation models to simulate the behavior of a complete system with regard to different physical and mechanical phenomena.

There are a variety of commercial and open-source simulation environments and platforms available for simulation of multibody models. In the commercial domain, there are popular simulation environments, for example *SimScape Multibody* (formerly known as SimMechanics) from MathWorks, *SIMPACK*, *Adams by MCS Software*, etc. These commercial software are general purpose and provide a possibility of modelling complex multibody systems. Apart from that, most of the commercial software provide a drag and drop approach to create systems using already available component libraries. They have standardized simulation models, which are good for a general simulation of a system to understand the dynamics but not for highly specialized products and they rarely provide any interconnection possibility between modules developed using different modelling tools.

3.1.1 SimScape Multibody

A popular commercial platform for mechanical multibody simulation modelling is SimScape Multibody (formerly known as SimMechanics) from MathWorks. SimScape provides a general-purpose environment for multibody modelling of complex systems such as robots, vehicle suspensions and construction equipment [9].

3.1.2 SIMPACK

SIMPACK is another commercially available general-purpose multibody simulation environment developed by the *Dassault Systemes.* It is aimed at mechanical system engineers for analysis and design of mechanical as well as mechatronic systems [10]. It offers domain specific support with add-on modules, for example *SIMPACK Automotive*.

3.1.3 SimulationX

Multiphysics simulation software SimulationX from ESI ITI GmbH is used for designing and modelling complex mechatronics systems and analyzing their behavior. It is used in many fields including automotive and provides specialized support and special model libraries for each physical domain [11]. It supports *Modelica* modelling language and could be integrated with other popular multibody simulation platforms using interfaces.

3.1.4 MBDyn

MBDyn is an open source general-purpose multibody dynamics analysis software developed by *Politecnico di Milano,* Italy. It includes support for integrated simulation of multidisciplinary Multiphysics systems. It has community-based support for development and further releases. It could be used along with other popular modelling programs such as MATLAB using C/C++ or Python based APIs. It is an analysis software and lacks a graphical user interface (GUI) and any modelling support. It is a command line tool targeted to academic research [12].

3.1.5 DyFaSim

DyFaSim (Dynamische Fahrzeug Simulation; English: *Dynamic Vehicle Simulation*) is another house-project of LuK, developed by the Simulation Development department. It provides the possibility of creating dynamic multibody simulation models with the help of standard simulation elements and components. It also provides company-wide standard modelling environment to fulfill the needs without any license costs. Unlike CluSys, DyFaSim has its own solvers as well as supports external solvers. It is fully compatible with CluSys and other standard software used in the company. The models created in DyFaSim are integrated in CluSys.

3.2 Tool Independent Integrated Simulation Platforms

The above examples show different modelling environments suitable for multibody simulation models. These environments, more or less, have the same general-purpose characteristics and provide limited support for tool-independent chain simulation processes. These solutions are only feasible if all the simulation models are developed using one modelling tool. The problem arises when simulation models and subsystems are created using different CAE (Computer Aided Engineering) and modelling software and are multidisciplinary in nature. For example, in clutch systems simulation and analysis, engineers create different mathematical models in different departments of the company and these models are used for an overall system analysis and product optimization based on the results. Sometimes, even a specific simulation module could be provided by an external company. Following are two examples of such platforms that provide tool-independent integration of simulation modules into systems.

3.2.1 Model.CONNECT

Model.CONNECT is a rather new solution for tool independent integration of simulation models in one platform. It is developed by *AVL* to manage the complex simulation development and analysis process in automobile industry. It targets the heterogeneity of simulation tool landscape in automotive industry and provides a unified platform to integrate models created in different simulation tools. It provides a unique blend of virtual as well as real components to facilitate the vehicle development process [13]. It is a new platform launched in April 2016 with few of its patents still pending approval [13]. Model.CONNECT provides a SIMULINK style drag-and-drop environment for integrating different simulation models and components forming the complete system. The integration is done through various standardized interfaces for popular simulation tools as well as for Functional Mockup Units (FMU). The intercommunication between different simulation models is done via *ports.* User defines these ports between parameters of different models manually by drawing lines between them. Once the models are connected, the simulation could be run and the results could be visualized in real-time as well as stored for later analysis.

In Model.CONNECT there is no mechanism for recognizing the parameters defined in simulations models and connecting them automatically to enable the data exchange.

3.2.2 EuroSim

EuroSim is a simulation platform that is developed by the Netherland Space Office (NSO) in cooperation with European Space Agency (ESA). It was originally developed for European Robotic Arm (ERA) simulation and since been used by many companies in space programs and in defense domain for flight simulators and as a development and test environment. It is managed by a consortium consisting of various companies [14]. The main purpose of EuroSim is to reuse already developed simulation modules in a unified environment enhancing the portability of modules. It takes care about the complete simulation package along with its configuration with respect to other simulation models. In general, EuroSim is different to other off-the-shelf solutions, as it is developed to provide an environment for creating chain simulations, rather than a modelling environment. It resembles CluSys in its functionality and purpose, but it is designed as a domain-specific tool for space and flight simulators. For model integration, EuroSim uses a *data pool* and *parameter exchange file.* This approach of data exchange is described in the following paragraph.

Data pool and Parameter Exchange File

EuroSim uses a complex approach for data exchange between simulation models. It allows the user to define the simulation parameters in a *Model Description Editor*. These parameters would then be collected during the runtime creating a so-called *"data-pool"*, the values of these parameters in the data-pool are then exchanged as defined in a separate *Parameter Exchange File.* This exchange file works as a configuration file for the simulation model. Before a simulation model is run, the data is exchanged and updated using the data-pool [15].

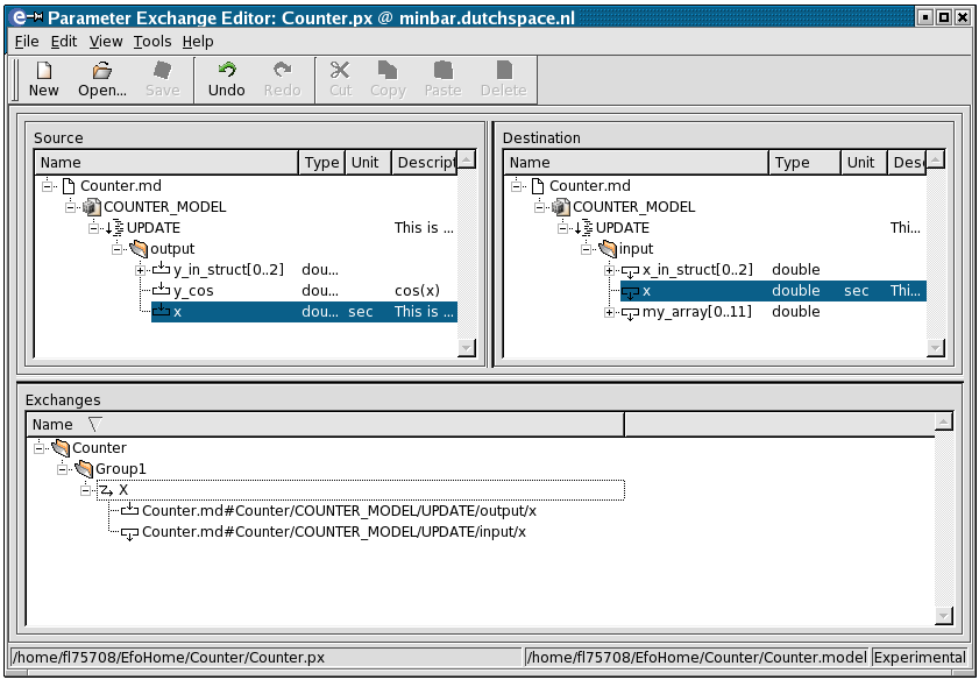


Figure 9: EuroSim Parameter Exchange File editor [15]

In Figure 9, an example of a *parameter exchange file* could be seen. In the editor, source and destination parameters for different modules could be defined. In the above example, a parameter *x* from a model is defined as output parameter in the left pane of the window, whereas the same parameter is defined as an input parameter for another model in the right pane.

The above solution is model centric, because the EuroSim platform is used for simulation models in different domains. The parameter exchange file makes it possible to save all the information needed for data exchange for a certain module combination and it could be ported along with the model.

3.3 State of the Art in CluSys

Simulation development department at LuK has already developed the simulation platform CluSys, where different simulation models could be integrated as calculation kernels and chosen to create chain simulations. This custom-made solution is important for the company as it allows the standardized product development, as well as maintains the confidentiality of the product in question. With off-the-shelf solutions, it is still required to transform them to meet company-specific requirements. In most cases, it is hard to integrate different simulation models from different domain specific CAE software into one complete system, because these commercial software come with their own propriety file types and data

systems. This requires extra effort apart from the high license costs. Apart from these standard commercial solutions, simulation development department also has a house-developed MBS software DyFaSim. DyFaSim currently does not support any standardized modelling approach or language like *Modelica*, so it is harder to integrate DyFaSim models in off-the-shelf solutions. CluSys, on the other hand, supports full integration of DyFaSim models.

3.4.3 Current Solution

In CluSys, simulation modules with different physical and mathematical models can interact with each other. These models are developed using different simulation or CAE software and CluSys provides a unified environment using a range of standard interfaces for intercommunication between the underlying calculation kernels.

CluSys already has a possibility of defining data exchange between different simulation modules (in the form of calculation kernels) using GUI (graphical user interface). For this purpose, CluSys generates lists of possible inputs and outputs (Figure 10) for all simulation modules in the toolchain. These lists are based on the defined parameters by the simulation experts, who, then go through all the parameters using the current GUI, to define data exchange between parameters. In Figure 10 and Figure 11, the source (left) parameters and target (right) parameters can be seen in the dialogs in unconnected and connected states respectively. Due to the complexity of simulation models of these subsystems, these parameter lists could be very large and comprise of hundreds of parameters. In the simulation toolchain, these modules are calculated sequentially and the input and output parameters of one module are available as possible input to the next module. This makes these lists even larger and sometimes there are over five hundred different parameters available in a single data assignment list.

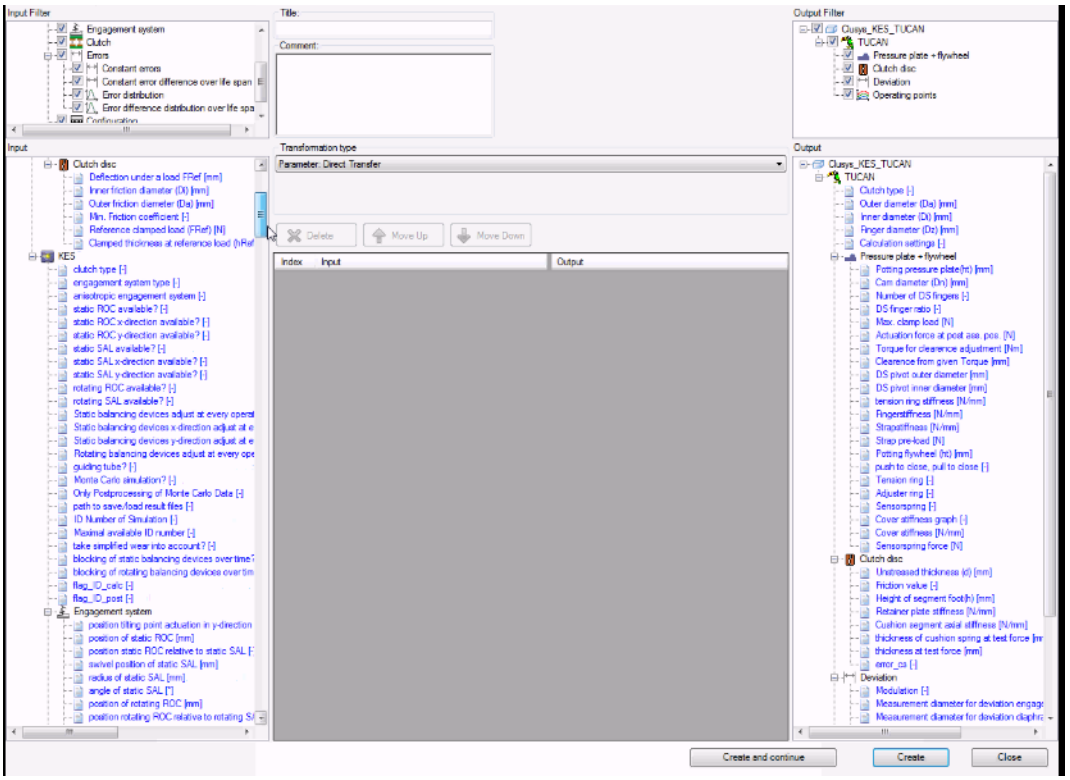


Figure 10: Data assignment dialog with lists of input and output parameters in modules

This intercommunication is done manually by the experts for each simulation project. In order to connect two different calculation kernels, experts have to evaluate the input and output parameters of both modules to identify if they represent the same physical quantity or value. These intercommunications between the modules sometimes require appropriate data adaptation called the *data transformations.* These data transformations are also defined by experts based on the requirements of the input parameter before the data is transferred.

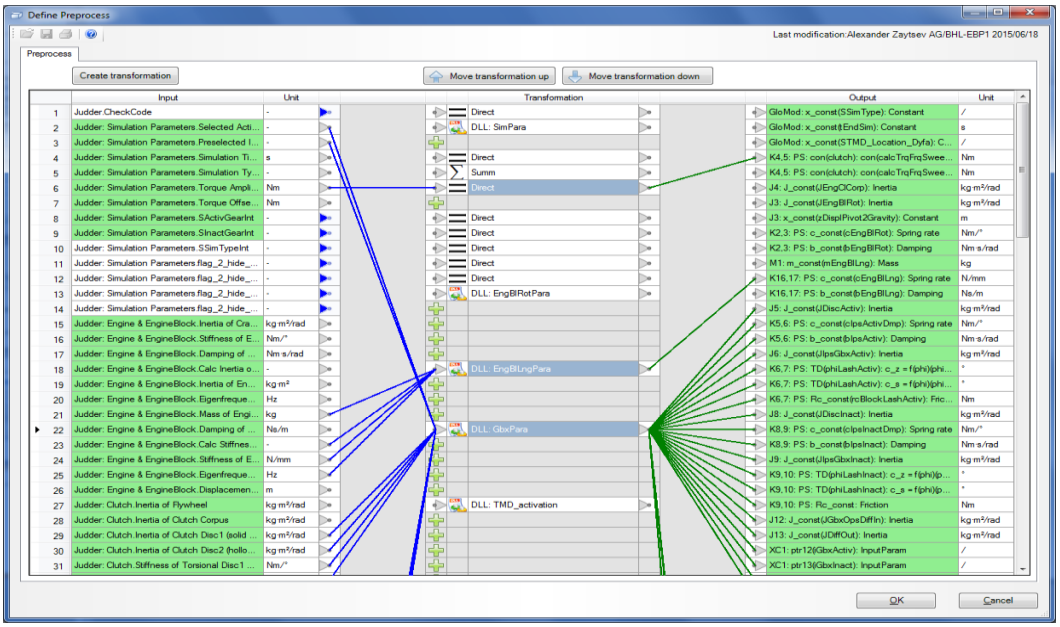


Figure 11: Defined data assignments between parameters of different modules

3.3.1 Data Transformation

The simulation modules are developed in different simulation software, for example MATLAB, DyFaSim or SimulationX. The simulation platform CluSys is connected to these calculation kernels via appropriate interfaces. CluSys takes the values of the parameters from the user and the calculation or simulation is run on the calculation kernels. The processed output data is then represented in the CluSys. In a simulation toolchain, this processed data is fed to the next module for further processing. A *data transformation* between the parameters of two simulation modules transforms or adapts the data in such a way that the value from one parameter could be used by another. For example, a *transformation* takes care about the unit of the output parameter from one module and converts it to appropriate scale, or changes the value entirely based on a predefined mathematical function so that the value is readable and within the range of the input parameter of the other module.

3.4.4 Disadvantages of the Current Solution

These simulation modules are developed by engineers in their respective departments. Referring to the example of judder in previous chapter (0), it could be seen that these simulation modules have different calculation results. A clutch system module, for example, Load Adjustment Clutch (LACK1), has clutch components, such as a clutch disc, with parameters defining its geometry, stiffness properties and friction

coefficients of the contact surfaces. This module simulates the behavior of a clutch system. The module that calculates the torque fluctuation during engagement of a clutch system (TUCAN) also requires the values of parameters of the clutch disc. Both these modules have a mathematical model of a clutch disc with similar parameters. These models are developed by different experts (see Figure 12). In the chain simulation, these modules intercommunicate and the torque fluctuations are calculated based on the specified values for the clutch parameters. For that intercommunication, these parameters must be interconnected, so that the values specified for clutch properties in the clutch module could be transferred to the torque-calculating module.

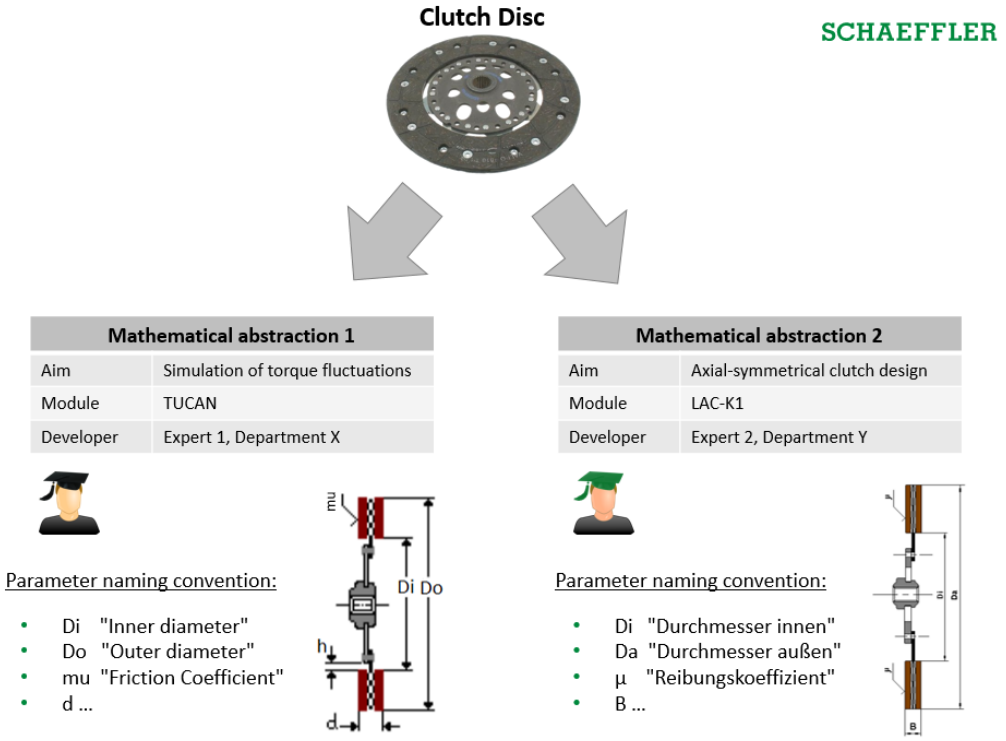


Figure 12: Mathematical representations of a single object in different simulation modules

In Figure 12, two mathematical representations of a single physical object can be seen with different parameter names and labels. Due to the lack of a unified naming convention, experts are free to use any label for a parameter. In above example one expert has used “Do” to represent “outer diameter”, while the other expert has used “Da” to represent “außen Durchmesser” (German: outer diameter), which is exactly the same physical property. These ambiguous or non-descriptive names for parameters representing the same physical quantity make it even more difficult to connect the right parameters just by seeing the names in the GUI. The user needs additional information to decide about the correct data exchange. This extra

knowledge about the parameters cannot be guaranteed, taking into account the fact that it could be developed by a different expert with different expertise in a different field.

There is a possibility of mistakenly connecting two parameters because of ambiguous naming convention and running the simulation with incorrect data. Additionally, there is also a possibility of error due to the communication overhead between the experts.

Furthermore, if the user wants to change one module in the simulation toolchain, all the previous intercommunications between the modules are lost, even if the new module has exactly the same parameters. The intercommunication has to be defined again manually before running the simulation again.

Apart from consuming more time, this approach could cause many parameterization errors, due to the complexity of modern simulation models and the large number of parameters.

Due to the manual intercommunication, only expert users are able to create the simulation toolchains and product engineers (end users) can only use the pre-defined tool chains. This approach results in lower acceptance of the simulation platform in product development department.

Summary

The commercially available simulation platforms are general purpose and they provide users with a range of modelling possibilities with off-the-shelf objects and components. Most of these platform and software packages support general purpose as well as domain specific multibody modelling but not the interconnection, integration and data exchange between these models to simulate complete systems. Above comparison of different off-the-shelf simulation and modelling software as well as available solutions for a unified simulation platform gives an idea about the state of the art in the tool-independent simulation market. The benefits of having a custom-made solution are far more than using any of the other mentioned solutions. Fortunately, we already have a highly customized, fully integrated simulation platform, which allows the simulation experts to define data processes between the simulation modules before the simulation is run. This intercommunication process is manual and requires a lot of time, prone to parametrization errors, demands expert-level knowledge about simulation models and makes the simulation environment unattractive for end user. The purpose of this master thesis is to automate this data exchange.

Chapter 4

4. Analysis and Requirements

In this chapter, the previously explained solutions and approaches for intercommunication between the simulation modules in a chain simulation process will be analyzed. As discussed in previous chapter, the already available manual intercommunication mechanism in CluSys is time consuming and could result in parametrization errors. Due to complexity of modern simulation toolchains and number of simulation modules used in a chain simulation, the graphical user interface for manual intercommunication between modules becomes more and more complex and difficult to use. Apart from the difficulty, manual method of defining data transfer makes it impossible to test and analyze different module combinations for a given problem.

To automate the intercommunication between simulation modules and the underlying calculation kernels in CluSys, the data exchange process between the individual parameters of these simulation modules has to be automated. For this automation, extra information about the individual parameters has to be given along with the modules. From the above chapter, it could be deduced that there are only a few solutions that cater the tool-independent module integration problem. The approaches used in tool-independent module integration in simulation environments will be analyzed with respect to the current situation in CluSys.

4.1 Configuration File Approach

One of the approach that could be used is similar to the *EuroSim* software discussed above. The manually defined intercommunication between modules is saved as a *Parameter Exchange File* and used like a configuration file whenever the project is loaded. This exchange file has to be saved with each project. This approach could automate the manual data assignment process for a defined project, but could not help if a single module is changed in the same project. In CluSys, a new functionality has been introduced that allows experts to define multiple module *replacements* in a single project (discussed in section 2.3.3). With this functionality, the end user has a liberty of choosing different modules and running the simulation to compare the results. The user does not has to define a complete new project for a different combination. This functionality could only work if the automatic intercommunication and data exchange is independent of the project or the module combination. A configuration file for a defined project will make the data

assignment process faster, but it will not work if user chooses to make any change in the same project.

4.2 Abstract Interface Approach

Another approach is to abstract the data exchange between different modules by creating an interface between the modules. This interface will recognize the parameters based on their physical quantity, regardless of the modules and interconnect them. This data exchange will be based on the fact, that two parameters from different modules have the same physical quantity and therefore the values of these parameters could be transferred. For that purpose, the interface could use a *dictionary* that defines the underlying physical meaning of a parameter. For example, if two modules in simulation toolchain project have a parameter that represents diameter of a clutch disc, the interface will interconnect them and the value from the first module will be used in the subsequent modules. Once the interface is developed, the user has to define the parameters of simulation modules in the database and they will be recognized by the interface automatically. Because of the abstract approach, this solution will also work with already developed simulation modules, as well as with any new modules.

The first approach is module-centric and could not supports module replacement functionality. It is easier to implement but fails to support flexibility in chain simulations. The second approach makes the data exchange independent of the parameter name, module name or module combination in a project, hence abstract enough to support module replacement. This approach also reduces the redundancy of defining configuration files for each simulation project. Each time the modules are used in a simulation project, their parameters will be recognized and data exchange will be defined automatically. This solution is intelligent, flexible and easily extendable.

4.3 Requirements

From the above analysis, it is clear that an abstract interface approach is better for the automation of data exchange in CluSys. To implement this approach, a database application to maintain a database of parameters containing their physical meaning has to be developed. This database will serve as a *parameter dictionary* and will be used by the *Automatic Intercommunication Interface.*

4.3.1 Intercommunication Database (parameter dictionary)

The requirements for the intercommunication database application are:

1. The database for the intercommunication interface should be capable of being accessed simultaneously, so that the multiple users from multiple CluSys instances could access the database all the time.
2. For maintaining the intercommunication database, a GUI application should be developed within the CluSys platform. The application should use already developed CluSys GUI elements and templates to increase the integration and provide user a seamless experience.
3. The application should has a possibility to add, delete and edit the entries in the database.
4. Before any change in the database, the application should check if the changes are valid and allowed.
5. There should be search and filter options in the dialog to facilitate the user in maintaining and extending the database.

4.3.2 Automatic Intercommunication Interface

For intercommunication interface, the requirements are as follows:

1. The automatic interface should use the intercommunication database for recognizing the parameters of the simulation modules in a project.
2. Based on the recognized physical quantity of the parameters, interface should create intercommunication between parameters if they have the same physical quantity. Interface should define the appropriate data adaptation or "*data transformations*" while creating the intercommunication between the modules.
3. Interface should be able to create intercommunication, every time a module is replaced or added in the project.
4. Interface should be able to recognize the conflicts, if there are multiple choices for intercommunication between source and destination parameters. Conflicts should be resolved by the user manually.
5. An intuitive way of resolving the conflicts should be implemented and the user decisions about the conflicts should be saved for future reference.
6. Interface and the conflict resolution mechanism should be integrated in the main CluSys platform and work parallel to the already existing manual interface.

Summary

Due to complexity of modern simulation toolchains, the manual intercommunication user interface is inefficient and prone to errors. To automate this process, additional information about the parameters in simulation models is needed. The configuration file approach, which is used by the EuroSim simulation environment, is module-specific and not abstract enough to be implemented in CluSys. The interface approach could solve the problem and is independent of the module or module combination in a simulation project and could support the module replacement functionality. To implement the interface approach, a database containing information about the parameters in simulation models has to be developed.

The main requirements for the database are a possibility to manage it using easy GUI, simultaneous accessibility and data integrity. The main requirement for the intercommunication interface is that the intercommunication logic should be based on the physical meaning behind a parameter and therefore should use the *parameter dictionary*.

Chapter 5

5. Programming Environment and Technology

This chapter covers the introduction about the programming environment and the structure of the simulation platform CluSys. First, the working environment and technology will be introduced briefly. Then the overall modular class structure of CluSys is explained briefly. The important member functions of each class and the intercommunication between the classes is explained. Lastly, a short introduction to different database technologies would be given and a comparison is done to analyze the requirements for our project.

In the Simulation Development department and company-wide in general, Microsoft tools are widely used. Microsoft Visual Studio Professional 2013 is used for development, as it provides the support for .NET framework and easy integration with other Microsoft tools used in the development department. Apart from the programming language, there are specific coding conventions and standard libraries that are used in the Simulation Development department. These conventions will be mentioned as such, wherever it is required in the following sections.

5.1 .NET Framework

.NET framework is a Microsoft technology that enables the development of different applications in a stable and consistent object-oriented environment. It provides a possibility of safe execution code with greater interoperability within the languages using the framework. It takes care of different aspects [16]:

- Developers can use the language of their choice. The framework takes care of the intercommunication between different .NET languages and integration of the code.
- A developer can use the Class Library of the framework or a functionality developed by other programmers, if the language targets the .NET framework. This increases the developers' productivity.
- Another great aspect of the .NET framework is its memory management. It takes care automatically about the object references and releases the memory as the object goes out of scope.
- The interoperability between different languages makes it attractive for a project, like CluSys, that requires continuous maintaining and development. The old and new functionalities could still be used regardless of their programming language.

5.1.1 C++ and C++/CLI

The main advantage of using .NET framework for application development is its language independency and use of a single framework for implementing features like security, memory management and exception handling. The Common Language Runtime (clr) of .NET framework provides interoperability between different programming languages that use the framework. Managed code is a term used for the code that is written using the guidelines and specification of the Common Language Infrastructure. This code is first converted to a platform-independent intermediate code known as Microsoft Intermediate Language (MSIL). The MSIL is then converted by the CLR into processor specific machine code [17].

C++/CLI is the specification of C++ language created by Microsoft to enable the use of C++ language within the .NET framework. It is considered as a language in its own, unlikely to the extension of C++, which is known as Managed C++ (MC++). Both terms are used interchangeably for C++/CLI. It uses a slightly different set of keywords and syntax to avoid any confusion between native code and C++/CLI. By using C++/CLI, it is possible to have the advantage of .NET framework, as well as the team members do not have to learn a complete new set of syntax. With C++/CLI, there would be easy communication between different languages that use the .NET framework. During compilation of C++/CLI code, it could be decided not to use .NET framework and compile the code as a native C++ program.

5.2 CluSys Structure

The internal structure of CluSys has been changed with time since its conception. The current structure of the program is modular and based on object-oriented approach. The module containing the program logic called the *Object* is written in native C++ (unmanaged C++) and GUI module called the *Dialog* is developed in C++/CLI (Common Language Infrastructure, also known as managed C++). The two program modules have a strict division and are independent of each other. No program logic is defined directly in the Dialog module. The user inputs from dialogs are saved in a special data Class called *Settings*. All the information between Object and Dialog modules is exchanged through a third module known as *Mediator*. The Mediator has access to both the Object data structures and Settings. The exchange of information between the Mediator and the Dialog is done via a special interface called *Mediator-Dialog-Interface*, which translates native C++ code into

C++/CLI code [2]. Following figure (Figure 13) shows the modular structure of the CluSys platform.

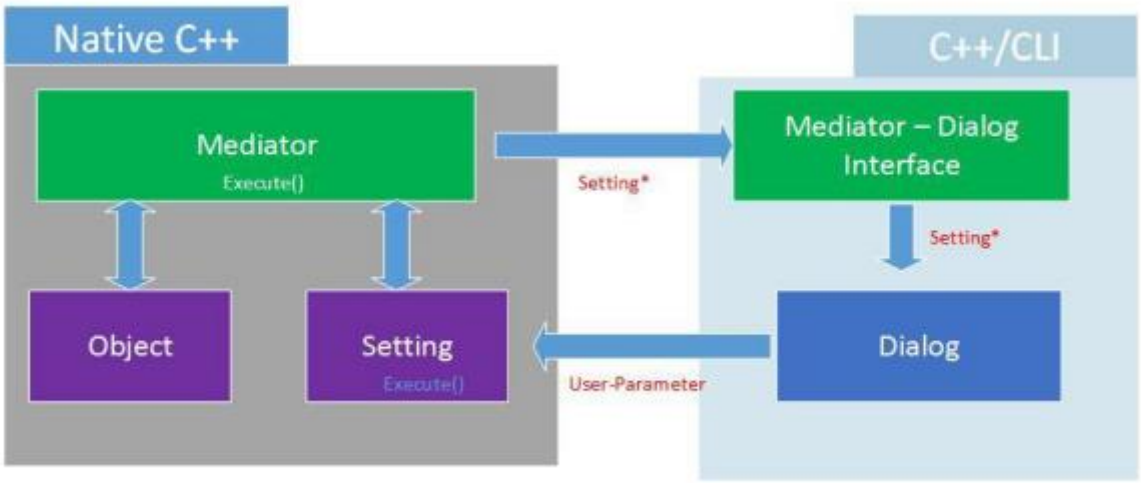


Figure 13: Modular structure of CluSys [18]

This complex modular structure makes the program logic clean and easy to expand without affecting other projects in the program. The code could be re-used for similar use cases. Replacing a specific module with a new one and with a better approach requires much less effort, especially in the case of upgrading the GUI of the program. Each of these modules would be explained briefly in the following lines.

5.2.1 General Functionalities

General Functionalities are the already developed functionalities and base classes that are used in the software development process to avoid extra effort and to facilitate department-wide integration. General Functionalities are accessible by not only CluSys, but are used in other projects and simulation programs (for example, DyFaSim) as well [2].

5.2.2 Object

The *Object* module contains all the program logic and is directly connected to the calculation kernels. It is responsible for data exchange with the calculation kernels and preparing the output for presentation before it could be shown to the user. It is also responsible for data exchange with the databases. *Object* prepares the search query for the database depending on the Dialog or user input.

5.2.3 Settings

Settings module is a special data class that is used as an exchange medium and storage container for the *Dialog*. All the input parameters from the user via GUI are saved in *Settings* so that it could be accessed by the *Mediator* for data exchange. For each *Dialog*, a *Settings* data structure is created and it contains:

1. a *Dialog* identifier
2. a call-back function pointer
3. data exchange parameters

The callback function is a special function to call the specific *Mediator* with respect to the *Dialog*. The function is called with a specific value to initialize the `Execute` function in the *Mediator*.

5.2.4 Mediator

Mediator, as the name suggests, is the module that is placed in between the *Object* and the *Dialog* for data exchange. *Mediator* has access to both *Settings* (where the data from the dialogs is stored) and the *Object*. For example, for each sub project within the CluSys platform, a *Mediator* is created. Each *Mediator* has an `Execute` function, which is called by the dialog with a specific *action* number. The `Execute` function then calls the respective function in the *Object* module to undertake that specific action. An example of the `Execute` function is in the following code block.

```
int Mediator::Execute(int action)
{
      switch(action)
      {
            case Settings::Actions::A:
            {
                  Do_Action_A();
                  break;
            }
            case Settings::Actions::B:
            {
                  Do_Action_B();
                  break;
            }
      }
return 1;
}
```

The updated data is exchanged between *Object* and *Settings* by the *Mediator*. For that exchange, two functions are used:

`UpdateDlgSettingsFromObj:` This function updates the *Settings* data structure with information from the *Object* data structures.

`UpdateObjFromDlgSettings:` With the help of this function, the data from *Settings* is transferred to the *Object*.

These two functions are called very often to keep *Settings* and *Object* data structures up to date with the current data.

5.2.5 Dialog

The *Dialog* module implements the GUI. In CluSys, the dialogs are written in C++/CLI (managed C++) to make use of the .NET technology for better GUI experience for the user. To enable the division between the *Object* and *Dialog*, no program logic is undertaken in the *Dialog* itself. They function as a window to represent the processed data. This approach makes it possible to easily update or change the GUI-technology without damaging or affecting the program logic. *Dialog* can access the *Settings* component and the exchange of information is done through the *Mediator* via *Mediator-Dialog-Interface* (see Figure 13). This interface could process both Native C++ code and .NET C++/CLI code and work as a bridge between both technologies.

There are two important functions in *Dialog* module:

UpdateDlgSettingsFromDlg: This function updates the *Settings* module according to the user-entered information in the GUI.

UpdateDlgFromDlgSettings: This function is used to update the *Dialog* or GUI with processed information from the *Settings* module.

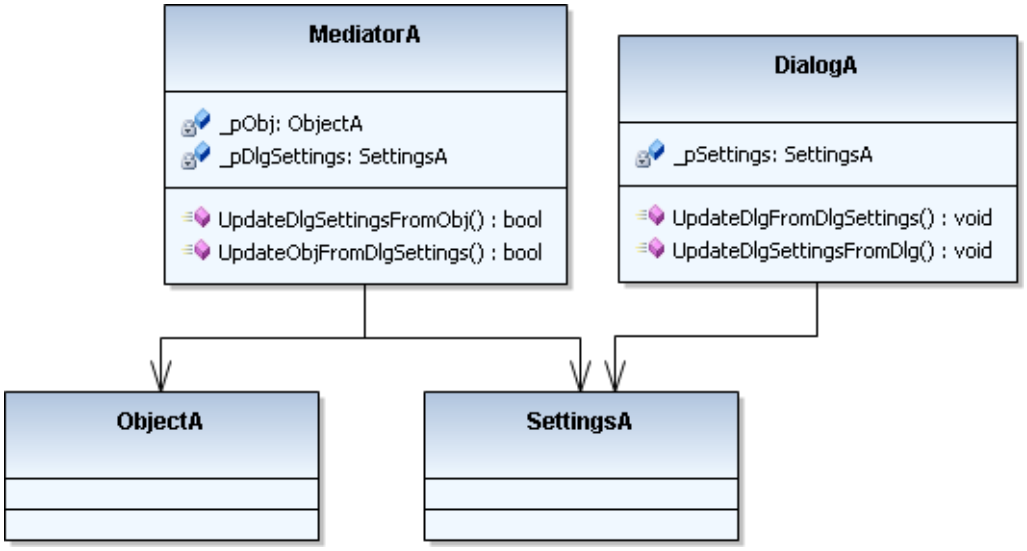


Figure 14: An example of Object-Dialog communication [2]

5.3 Object-Dialog Communication Cycle

The above figure (Figure 14) shows a generic example of the communication between *Object* and *Dialog* classes. At first the *Mediator* will call the function `UpdateDlgSettingsFromObj()` which will update the *Settings* with data from *Object*. Then the *Dialog* would call the `UpdateDlgFromDlgSettings()` which will get the data from *Settings* data structures and fill the dialog with it. After each event in the GUI, `UpdateDlgSettingsFromDlg()` will be called to update the *Settings* with user-defined input and the call-back function of the *Mediator* will be called with a specific action argument. The *Mediator* will call `UpdateObjFronDlgSettings()` to update the *Object* with current data and *Execute* function would be called with the specified *action*. After the *action* is completed, the *Mediator* will again call the `UpdateDlgSettingsFromObj()` to update the *Settings* data structures and the cycle would be repeated in the same order.

5.4 Database Technologies

To create an automated data exchange system between the simulation modules, the first step is to establish a database of parameters in such a way that it could be used by the automatic interface for recognizing the parameters with respect to their physical meaning. This database of parameters will work as a *dictionary* for the interface. There are multiple technologies available for databases and the decision of using one technology instead of any other lies in the fact that which type of database system would be suitable for the respective data. Most of the database management systems (DBMS) are based on two main types of databases – *relational (SQL) and NoSQL databases*. These types are described briefly in the following section and then a comparison is made.

5.4.1 Relational (SQL) Databases

In relational databases, data is organized using precise mathematical model, which enables the structural integrity and consistency and allows saving and managing the data without redundancy.

In a relational database model, data is represented in tables, which are called relations as they contain collection of the data of the same type [19]. The rows in a relational database table are always unique. To ensure the integrity of the data and prevent duplicate entries, a unique *identifier* is defined for each row. The *identifier* could be one or a group of columns in a row. This *identifier* is called a *primary key*.

Relational databases have a distinguishing property of *joining* two distinct tables to create a unique set of information that combines the data from two or more tables. For that purpose, a column containing the *primary key* values of one table should be present in the other table. This column is known as a *foreign key*. A *foreign key* value must be same as an existing *primary key* value. This ensures the *referential integrity* of the data and prevents any invalid or empty references [19]. This also restricts deleting any *primary key* that is referenced somewhere else as a *foreign key*. These conditions for a *foreign key* are called *foreign key constraints.*

To create a relationship between two different data tables, an entity is defined.

"An entity is something about which we store data." [19]

There are three basic ways of creating a *relationship* between two different entities, *One-to-One, One-to-Many* and *Many-to-Many.* All of these relationship types could be defined while creating a database. It is possible to have an instance of an entity without referencing it in a relationship. If it is mandatory for an entity to be in a

relationship, then it is referred as a *weak entity*. A weak entity cannot exist without a *mandatory relationship* to another entity. These concepts of a relational database assure the concurrency and integrity of the data all the time. Almost all of the relational databases use the Structured Query Language (SQL), which has a huge support and many established commercial and open-source solutions, one of which is MySQL.

5.4.2 NoSQL Databases

NoSQL databases are considered the opposite of relational databases, as they lack a fix structure and based on a horizontally scalable data model. The name NoSQL stands for ***N**ot **O**nly **SQL*** or *Non-SQL*. There is no standard database query language or rule therefore, the programmer is required to take care of all the implementation himself. All the NoSQL databases lack a defined schema and therefore considered simpler and flexible in their structure. The main highlights of a NoSQL database are enhanced availability, speed and scalability in case of large amount of data. For these reasons, NoSQL databases are widely used in distributed and web applications. The data in a NoSQL database is independent of each other and could be stored in many forms. The most popular types are *Key-Value*, *Document-oriented*, *Graph-oriented* and *Column-based* databases. [19]

Key-Value database:

These are the simplest form of NoSQL databases and data in these databases is associated with the help of “hash-tables” or “hash-maps”. A unique *key* is associated to a *value* and the key is used to locate the data in the database. As opposed to relational databases, a key in key-value database could be associated to any type of data.

Document-oriented database:

In a document-oriented form of NoSQL databases data is stored as a *document* instead of a relation (table) [20]. Each document has an identifier and could has a different structure. Within a document, data could be stored as in *key-value* database. There is no schema for the database and therefore the structure could be changed anytime without any prior requirement. These type of databases are suitable for large amount of unstructured data or in cases where the type of data

cannot be defined beforehand. A popular implementation of this type of NoSQL databases is open-source MongoDB.

Graph-oriented database:

Graph databases are an addition to the *document-based* databases for more rapid traversing of the data. Different *documents* are connected to each other creating a network, where each *document* represents a node. These data stores are widely used nowadays in social media websites.

Column-based database:

These type of NoSQL databases use a very different principle. The data is stored in the form of columns instead of rows, as is the case in a relational database. Columns are loosely connected in the form of a "*column-family*". Column-based databases are useful in the scenarios where column operations are done on the data, for example in analysis or statistical programs.

5.5 Comparison of Database Technologies

The NoSQL database design lays more importance towards availability and flexibility over integrity and concurrency of the stored data. In case of relational databases, more importance is given to concurrency and integrity of the data.

A relational model is suitable if the data to be saved is structured and remain on a larger part in the same structure. The schema or the structure of the database has to be defined before the data could be added and the data could be only added if it is based on that structure.

In the case of *parameter dictionary*, the main importance is on data integrity and concurrency. A dictionary cannot have redundant or invalid entries. The structure of the data could be easily defined and all the entries will follow the same structure. All the parameters will be defined with respect to their physical quantity and will be related to each other, thus allowing the interface to recognize the connection between two distinct parameters. Because of the limited size of the database, performance is not an issue in this case. Availability could be compromised but the consistency of data cannot be compromised, as it will invalidate the data exchange, if *parameter dictionary* has invalid data.

All of these aspects require that a relational database model is used for *parameter dictionary*. In the following section, the concept for the *parameter dictionary* based on the relational database model will be explained in detail.

Summary

The program structure of CluSys is complex and modular. It is divided into different modules to ensure transparency and reusability of code. Each module is based on a *BASECLASS* that has some default members. These base classes are used in all the sub-projects of CluSys to implement different functionalities. These classes have a pre-defined and strict division and intercommunication mechanism, which has to be followed in every sub-project. To implement the data of parameters or a *parameters dictionary*, a relational database model will be used. As compared to the non-relational or *NoSQL* databases, a relational or an *SQL* based database ensures data integrity and concurrency. A dictionary cannot have invalid or has more than one entries defining a single parameter.

Chapter 6

6. Concept

In this chapter, the concept for meeting the previously mentioned requirements would be explained. This chapter is further divided into three main sections; first section will explain the concept for Intercommunication Database or *parameter dictionary*. The second section covers the concept for Intercommunication Interface and the third section explains the concept for resolving data conflicts that may arise during automatic data exchange. This concept will be used in actual implementation of the solution, which is explained in the next chapter.

6.1 Intercommunication Database

The Intercommunication Database (parameter dictionary) would be the backbone of the interface logic. This database should contain enough information about the parameters, so that the interface would be able to recognize a parameter based on its physical meaning and make an intelligent decision about the interconnection. This dictionary will have a list of possible "physical meanings" which will be given a unique ID. The parameters of the modules will be linked to their respective meaning with the help of unique identifiers. Based on the relational database model (section), two tables (relations) will be created. These tables will be connected to each other to achieve the desired functionality.

6.1.1 Connector ID Table

To create a database of parameters referring to their physical quantities, first a table with a collection of all physical quantities as well as dimensionless values should be created. These quantities would be given a unique identifier. This unique identifier, called the ***Connector ID,*** would be the *primary key* for the table**.** The connector ID would be used by the interface to detect the corresponding physical meaning. The table would also contain the name and physical type of the quantity. Both of which could be freely chosen by the experts to reflect the meaning of the quantity. This information will help the expert users to maintain the table. Following is an example of a collection of physical quantities with a unique *Connector ID* and the quantity type.

Connector ID	Quantity name	Quantity type
ID_0001	Clutch disc inner diameter	Length
ID_0002	Engagement travel	Length
ID_0003	Engagement force	Force
ID_0004	"…"	"…"

Table 2: An example of a collection of Connector IDs

In the first row (Table 2), *ID_001* represents a physical property "clutch disc inner diameter", which has a physical quantity type of "length". The second row in the table 1 shows another physical measurement "engagement travel", which is the distance travelled by the engagement system when the clutch is engaged or "pressed". This property also has the physical quantity type "length". This example shows us the importance of having a unique identifier, as both of the two properties have the same physical type, but they represent different physical properties.

Index	Label	Module	Type	ID X-Axis	ID Y-Axis
1	InnerDiameter	Module1	Scalar	ID_0001	-
2	result.MaxTravel	Module2	Scalar	ID_0002	.
3	result.Torque	Module1	Char. curve	ID_0003	ID_0004
4	InnerDiameter	Module3	Scalar	ID_0001	.
…	…	…	…	…	…

Table 3: Parameter catalog for intercommunication database

6.1.2 Parameter Catalog

For cataloging the parameters from the simulation modules, another table will be created. This table would contain the information about the parameters and refer to the first table to define their physical quantity. The column with *Connector ID* will serve as the *foreign key* for this table. The information from this table should be enough for the interface to recognize the parameter in the module.

i. Label and Module Name

The parameter label within a simulation module is unique but it is possible to have a parameter with the same name in two different modules. The label for a parameter is defined by the experts, who are free to enter any name. The simulation platform generates a "path" for a parameter that contains the *container* (see section 2.1.1) label, the module label and the parameter label, all concatenated together with a "." between them. For example:

```
Container1.Module1.parameter1
```

This unique *path* identifies a single parameter in the project. The interface would use only the module name and the parameter label to recognize a parameter within a simulation project. In this way, the intercommunication logic could be more abstract.

ii. Type

There are different types of parameters in a simulation module, for example scalar parameter, characteristic curve, a parameter matrix, etc. For the automatic intercommunication of simulation modules, two of the possible parameter types, namely scalar and characteristic curves, are considered. In table 3, the first two entries are scalar parameters while the third parameter represents a characteristic curve.

iii. ID (X-axis and Y-axis)

Scalar parameters could be represented by one physical quantity but for a characteristic curve, more information is required. A characteristic curve is a representation of more than one point creating a graph in two-dimensional space. Each axis of the curve represents a parameter with a physical quantity or a dimensionless value. The *Connector ID* for both axis should be entered in the table to identify the physical meaning behind the curve.

6.1.3 Relationship Between Parameter and Connector ID

Each entry in the parameter catalog should contain at least one *Connector ID, which* must refer to an existing entry in the Connector ID table. That means the parameter

would be the *weak entity* in this case and could not exist without creating a relationship with a Connector ID. For example, if we want to define a parameter, which represents inner diameter of a clutch disc, there must be an entry in the Connector ID table, which defines the physical quantity *clutch disc inner diameter* with a unique ID. This ID would be then entered for all the parameters that represent inner diameter of a clutch disc. Therefore, there is a *one-to-one mandatory relationship* between parameter and a Connector ID.

Apart from the *mandatory relationship*, there is an optional *many-to-many* relationship model between the two entities. A Connector ID could be a part of more than one parameter definitions in the catalog and a parameter definition could contain more than one Connector IDs. Latter would be the case for defining a characteristic curve, which must contain two Connector IDs for both dimensions. This relationship between the two entities is shown in the following *entity-relationship diagram* (Figure 15).

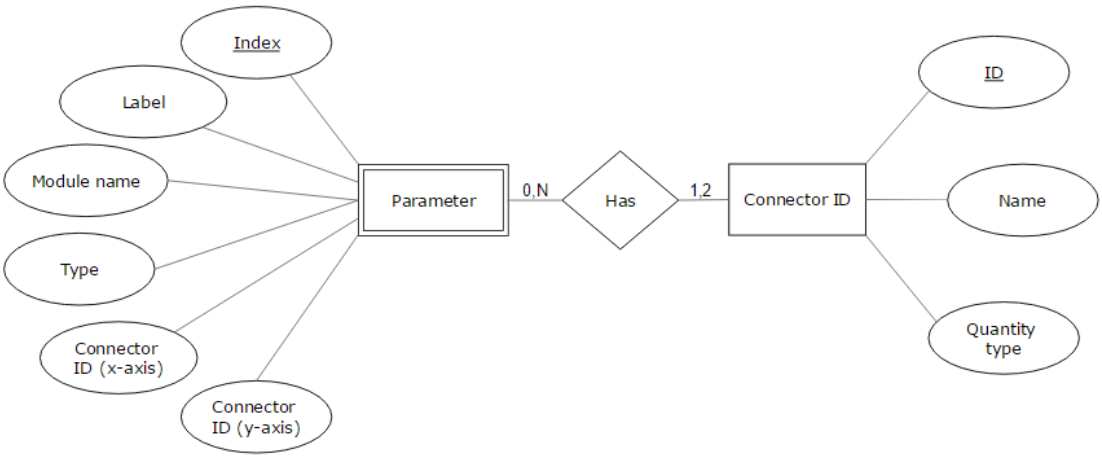


Figure 15: Entity-relationship diagram in Min-Max notation

In the above *entity-relationship* diagram, the two entities *parameter* and *Connector-ID* are shown in rectangular boxes with their attributes in ovals. The double outlined entity (parameter) is the *weak entity*. These entities have a *many-to-many* relationship with *cardinalities*[4] written in minimum and maximum notation [21]. *All* parameter entities "must have" at least *one* or "could have" at most *two* Connector IDs. One or

[4] In data modelling, cardinality is defined as the relationship of one table to another with respect to number of their elements.

at most two Connector IDs could be part of *n* number of parameters. The underlined attributes show that they are key values.

6.2 Kernel Intercommunication Interface

The second part of the concept is about the kernel intercommunication interface. In a simulation tool chain, as seen in the example shown in chapter 2 (see section 2.3.2), data is processed sequentially. The first module in the toolchain is calculated and the output from this module is made available to the next module.

The kernel intercommunication interface will create intercommunication between two modules in a simulation toolchain by interconnecting the input and output parameters and transferring the calculated values from one module to the next.

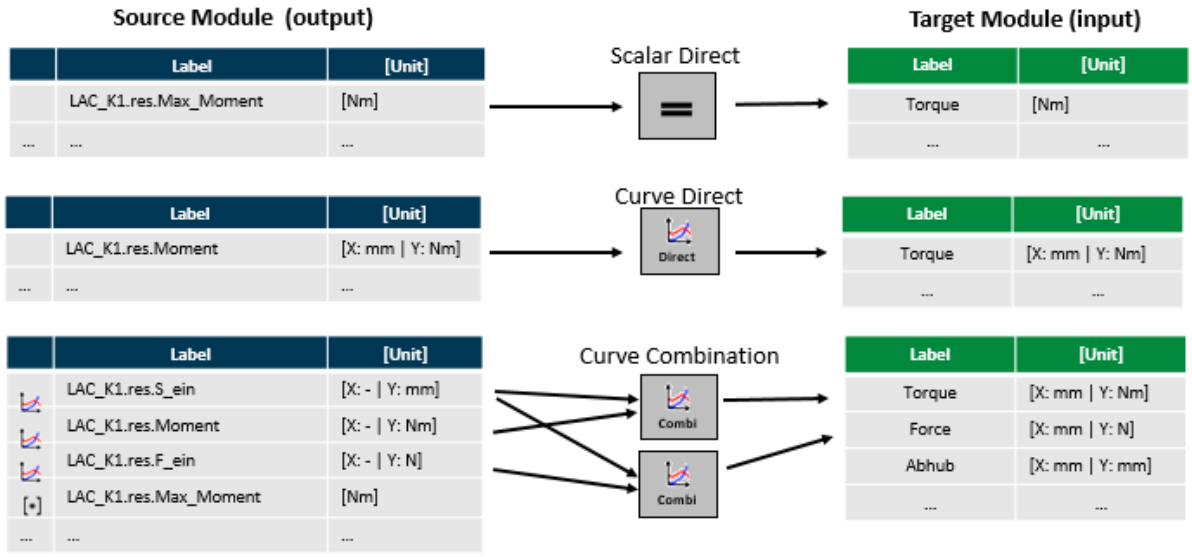


Figure 16: Intercommunication between two modules showing data transformations

Before the data is transferred to a module, it is needed to be adapted for the module in question. This adaptation is done through a so called "*data transformation*" (Figure 16). There is a range of different types of data *transformations* that are done within CluSys, depending on the *source* parameters and the requirements of the *destination* modules. Three types of transformations are in the scope of this thesis, these are:

- **Direct parameter transformation**

  This type of transformation directly transfers the value of the *source* parameter to the *target* parameter after converting the unit to the appropriate scale, if needed.

- **Direct curve transformation**

This transformation transfers the set of values representing the source curve directly to the *target* curve if both curves have same physical quantities as dimensions (Figure 17). The units are converted to the scale to match the *target* curve.

Figure 17: Direct curve transformation

- **Curve combination transformation**

This type of transformation takes two different *source* curves as input, combines their data and transfers to one *target* curve. For example, two *source* curves would be combined resulting in a new set of data points that could be used by a *target* curve for calculation. The requirement for this type of transformation is that both *source* curves should have one dimension matching the *target* curve and the other dimension must be common in both *source* curves. For example in Figure 18, *force* and *torque* dimensions will be combined to create a characteristic curve with the same dimensions as the *target* curve. Both *source* curves has *time* axis in common.

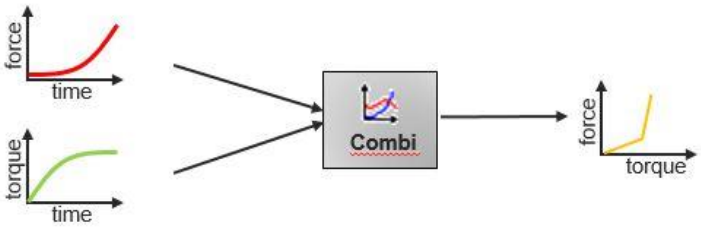


Figure 18: Curve combination transformation

The kernel intercommunication interface has to take care about the type of transformation required for all parameters based on their type.

6.2.1 Algorithm for Automatic Intercommunication

The interface (Figure 19) would recognize the destination or the *target* module and the *source* module and implement the following algorithm starting from the parameters in *target* module. The reason behind applying the algorithm to the target module is the restriction of assigning a target parameter only once. The source

parameters, on the other hand, could be assigned more than once to different targets. Going through the list of target parameters will assure that a target parameter is part of a data assignment only once.

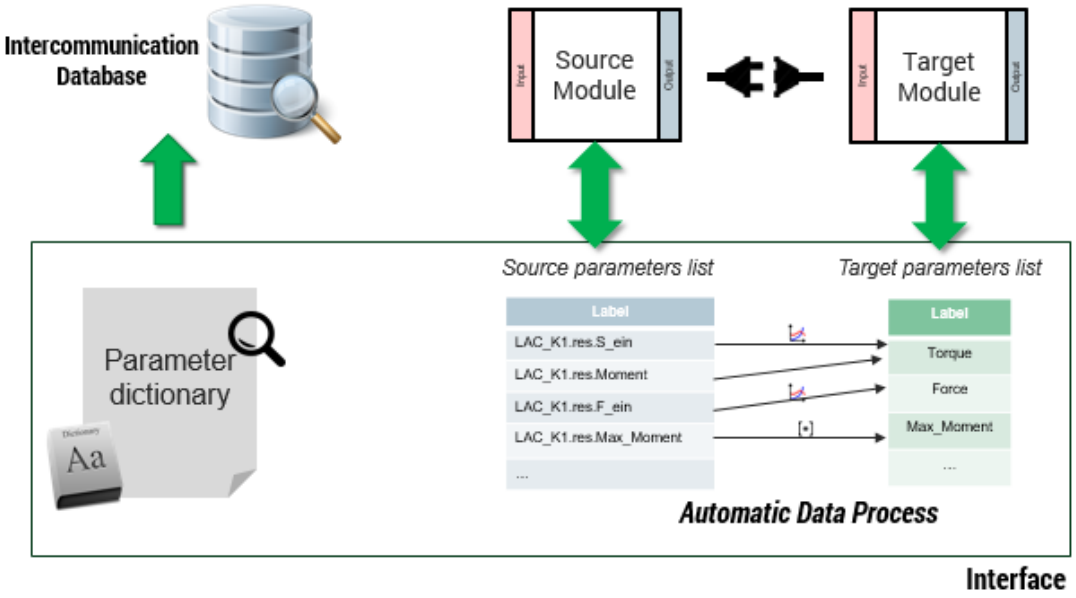


Figure 19: Automatic intercommunication interface

The basic algorithm can be seen in the flow chart diagram (Figure 20). Following is the pseudo-code for the algorithm in its simplest form:

```
for (all Target Module params)
      if (param → Exist in Intercommunication DB)
            for (all Source Module input params)
                  if (param → Exist in Intercommunication DB)
                        if (Connector ID == Connector ID)
                              → Connect!
```

In case of scalar parameters, the algorithm is same as above. For characteristic curves, first, the interface has to check the possibility of a curve which has same dimensions and whose set of data points could be *directly* transferred to the target curve. If no such curve is present in the *source* modules, then the interface will search for two *source* characteristic curves that could be *combined* with the help of a *combination transformation* to create a new set of data points for the *target* curve.

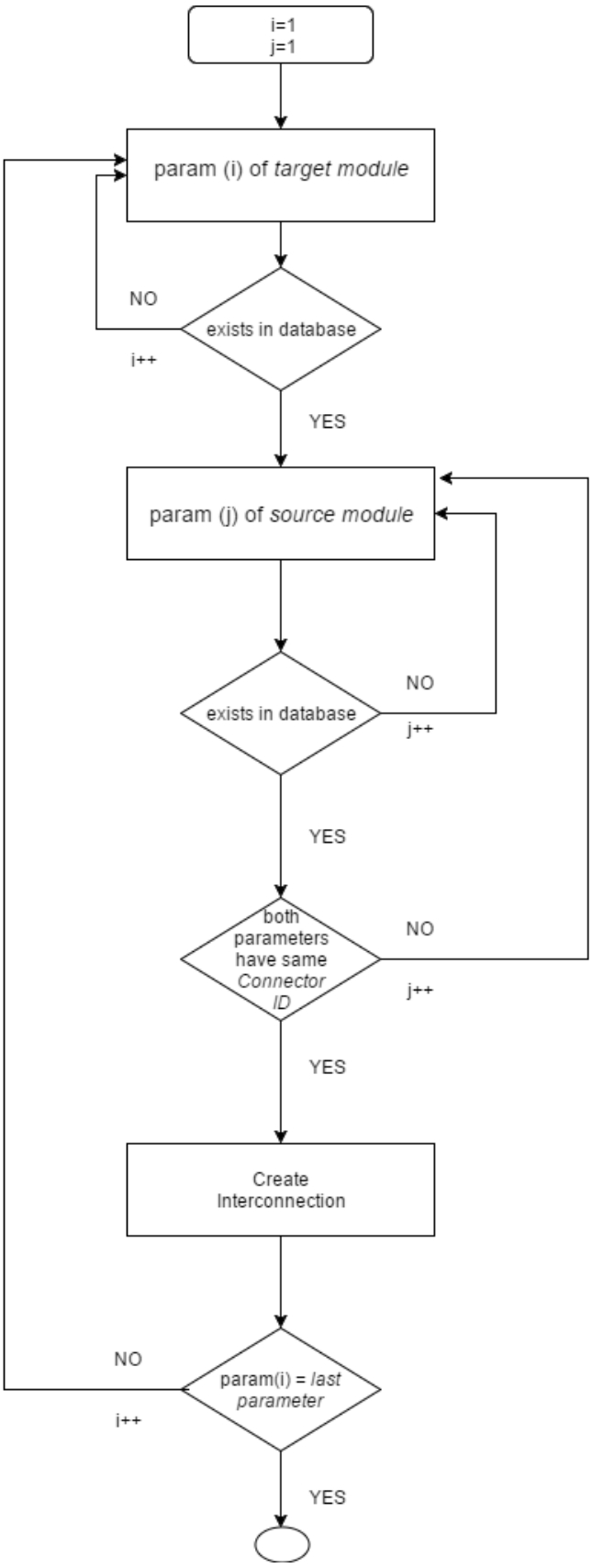


Figure 20: Flow chart of the basic intercommunication algorithm

The simplified algorithm for characteristic curves is as follows:

1. Get all characteristic curves in the target module
2. Go through all the curves and search if a characteristic curve is defined in *parameter dictionary*
3. If yes, recognize the dimension from the Connector IDs of both axis
4. Get all the curves present in the *source* list
5. Check if any of the curve exist is the *parameter dictionary*
6. If yes, get the Connector IDs of both axis
7. Check if the *target* curve and the *source* curve have same dimensions (that means same IDs for both axis)
8. If yes, create direct curve transformation
9. If no, check if the *source* curve has one axis that matches an axis of *target* curve
10. If yes (it is now curve A), search for the another curve (curve B) which has an axis matching the *target* curve
11. If found, check if the *Curve A* and *Curve B* have a common axis
12. If yes, define a curve combination transformation between the three curves

The above algorithms search through the *parameter dictionary* with the help of parameter *labels*, which are defined by the experts. The labels must be same in modules as well as in the dictionary. The physical meaning of the parameter is recognized by the interface using the Connector ID associated with the parameter. After a parameter is recognized by the interface, the interconnection logic only takes the Connector IDs that means the actual physical meaning behind the parameter, into account. That mechanism makes the intercommunication logic abstract and flexible enough to extend it further easily.

6.3 Data Conflicts

As seen in the example of a simulation tool chain in chapter 2, the number of simulation modules in a project could be more than two. The input and output parameters of previous modules are available as *source* to the next module.

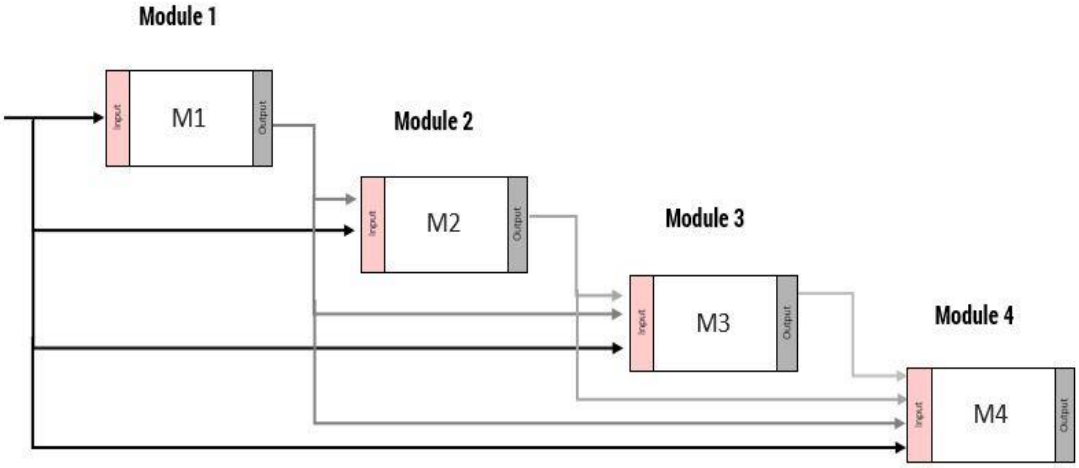


Figure 21: Number of possible inputs for subsequent modules

That means the number of possible *source* parameters for one *target* parameter would increase with each subsequent module. For example in Figure 21**,** it can be seen that there is a possibility of having more than one parameter with the same Connector ID in the *source* list for a certain *target* parameter in Module 3 and 4. This could cause a *data conflict*.

There could be following types of conflict situations:

- **Multiple sources - one target**
- **Multiple sources – multiple targets**

The possible conflicts can be seen in the Figure 22. In both of the above mention cases, the interface should create, a list of possible *source* parameters for a *target* and prompt the user to *resolve* the conflict by choosing one of the possible inputs. This *resolved* conflict would be stored for future reference. Interface would connect automatically the *target* parameter with the user-selected *source* parameter next time the same conflict situation arises.

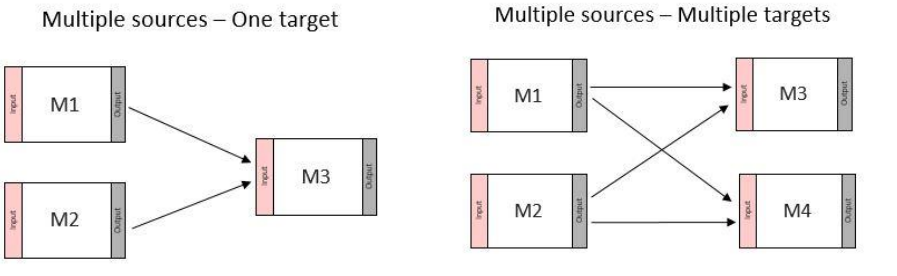


Figure 22: Possible data conflicts

6.3.1 Example of a Conflict

In the Figure 23, we have two modules, a *Double clutch assembly* and a *Lever Actuator*. These two modules are used in a project and the expert wants to interconnect

the parameters of both modules. The problem here is that the double clutch assembly has two different clutch parts. That means some of the parameters may be doubled. In manual intercommunication situation, the experts know which clutch part they want to interconnect, so they connect the parameters. The automatic interface could not decide for itself, as it will see that parameters from both parts are available in the *source* list and they correspond to the same physical quantity. In such situations, an expert should be prompted to choose from the available options.

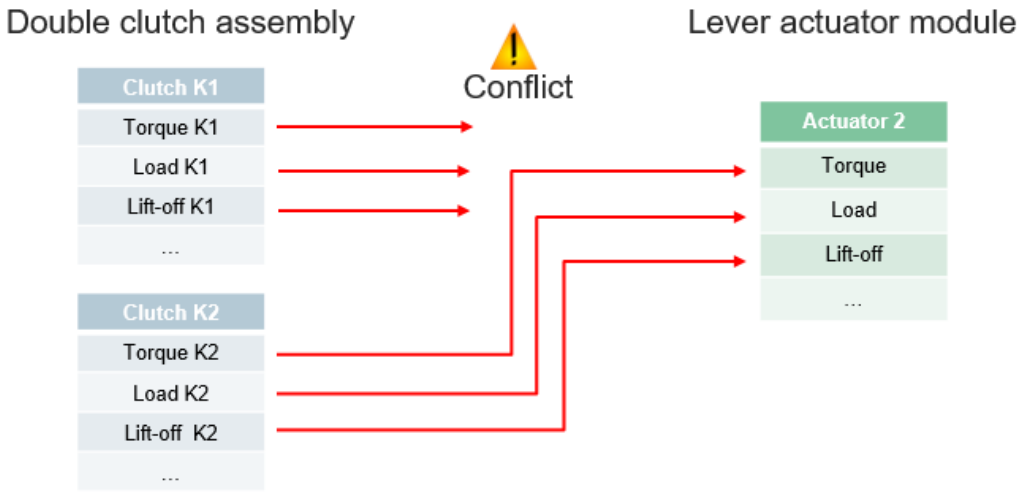


Figure 23: Example of a data conflict

6.3.2 Example of a False Conflict

In the Figure 24, a parameter from module 1 (torque) is connected to a parameter of same name in module 2. For module 3, both of these parameters would be available as possible source parameters. In this case, the torque from module 2 is getting the value of torque from module 1, which means the basic input for torque for this project is the value from module 1. The interface detects this situation and the original value from module 1 is used for all subsequent modules. This situation should not be saved as a data conflict.

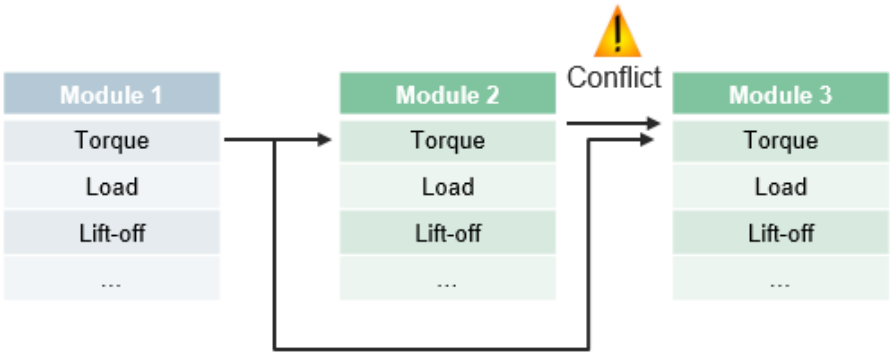


Figure 24: Example of a false conflict

Summary

The Automatic Intercommunication Interface will use the *parameter dictionary* to recognize the input and output parameters of simulation modules based on their physical meaning. It will interconnect two parameters if they both represent a same physical quantity. Interface logic would be independent of the modules in a project. This abstract and flexible nature is useful in extending the interface.

Chapter 7

7. Implementation

In this chapter, the implementation of the previously explained concepts will be discussed in detail. First, the implementation of different tasks will be explained in reference to the used technology and class diagram. The implementation of the intercommunication algorithm will be explained and a few important classes and functions will be described in detail. Lastly, the working of the Intercommunication Interface with the module "*replacement*" functionality will be explained.

7.1 Intercommunication Database (parameter dictionary)

As it is stated in the concept that the Intercommunication Interface will use a database that contains the parameters and their physical meaning with respect to the physical quantity or physical characteristic they represent. For this *parameter dictionary*, an application is created within the CluSys as a sub-project. All the sub-projects created in CluSys are inherited from a list of parent classes, so that the architecture of the overall program will remain the same and the intercommunication between different modules of the program could be made possible.

First, the GUI of the *parameter dictionary* application will be explained and then the class diagram of the *parameter dictionary* will be explained in detail.

7.1.1 Parameter Dictionary GUI

In Figure 25, the graphical user interface of the parameter dictionary is shown. The GUI of the application dialog is based on the OK-Cancel template of CluSys dialogs. To standardize the development and to reuse the already developed code, there are many templates available for CluSys development. CluSys developers use these templates to extend the functionalities, but sticking to the same basic dialog template. It is important to provide a seamless user experience throughout the environment. This is a non-functional requirement of the CluSys development. The application GUI is developed using .NET compatible C++/CLI and it used .NET Class Library [22] elements for GUI components. Each of the GUI component of the application will be explained briefly in the following:

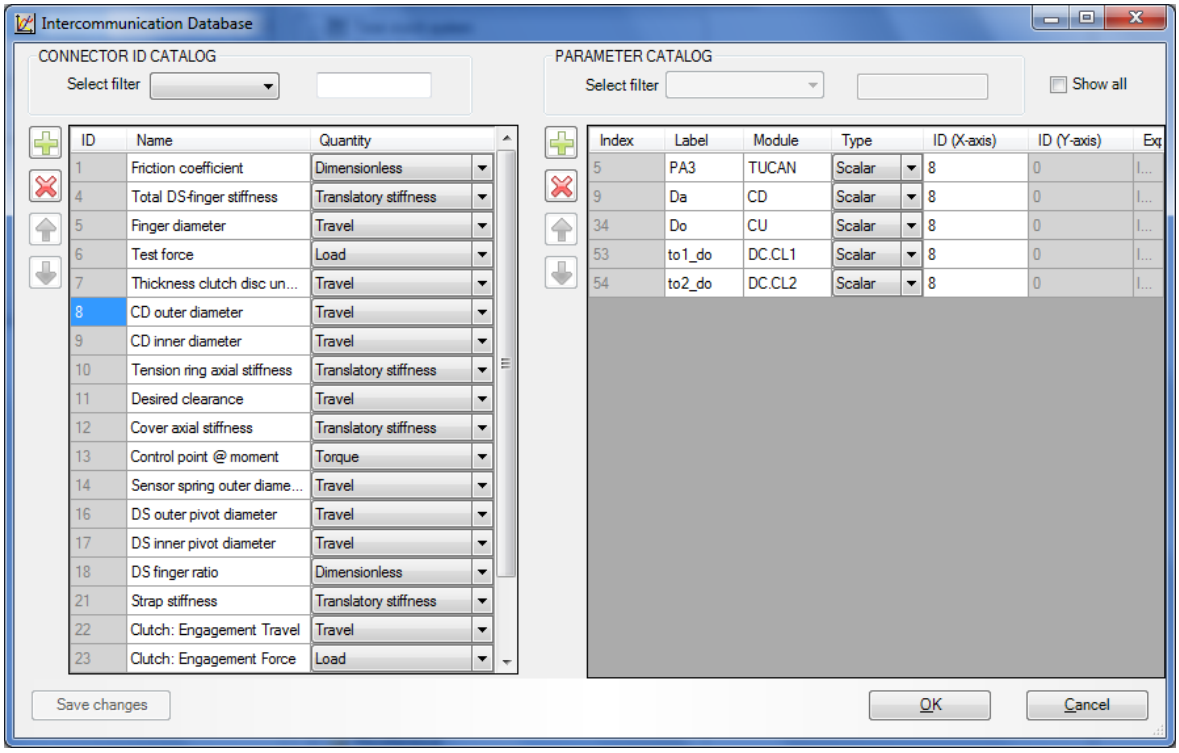


Figure 25: GUI of Intercommunication Database (parameter dictionary)

- **Connector ID Catalog**

The dialog of the parameter dictionary contains two tables for showing the Connector IDs and the parameters respectively. On the left side is the Connector ID catalog. This is implemented with the help of a *DataGridView* element of provided by the .NET Class Library and has three columns. It displays the data in a grid format for easy customization. First column *ID* contains the unique ID given to the physical quantities. This is a *read-only* column. Second column is *Name* and contains the name or labels of the quantities. User can input any string of alphabets, numbers or special characters to define the name of the quantity. Third column is *Quantity*, which contains a drop down list of CluSys-wide available quantity types. User can select a quantity type from the list. The quantity type "*Dimensionless*" is selected by default.

- **Parameter Catalog**

On the right side of the application dialog is the parameter catalog, which is also a *DataGridView*. The parameter catalog has seven columns to display different attributes of a parameter. First column shows the *Index* of the parameter. It is a

read-only column. Second column is *Label*, which displays the label of the parameter. Third column is *Module*, which is to define the module name of the respective parameter. As explained in the previous chapter (section 6.1.2), the parameter label and module name must be the same as defined in the calculation kernels or simulation modules. Fourth column is a drop down list for selection between the parameter *type*, that are, *scalar* or *characteristic curve.* The fifth and sixth columns are for entering the ID number of the quantity. They are labels as *ID (x-axis)* and *ID (y-axis)*. ID (x-axis) column is activated by default and is used to enter the ID of parameter's physical quantity, which must be defined before it could be entered in this column. This column is used to define the scalar parameters. In case *parameter type* is *Characteristic Curve*, the *ID (y-axis)* column is activated and physical quantities representing both axis of a characteristic curve could be entered in respective columns. The last column *Expert* is also a *read-only* column and shows the username of expert user who defined the parameter in the database.

- **Search and Filter Options**

Above these *DataGridView* elements, filter and search options can be seen. These options are useful in case the Connector Id and parameter lists are very long. A filter could be selected with the help of a drop down list containing the column headers. If a filter is selected, the search bar next to the filter will be activated and as the user, types in to the search bar, the entries in the respective table will be shown with respect to the search term entered in the bar.

- **Show All Parameters (check-box)**

There is a *check-box* element in the upper right corner of the dialog. If this check box is 'checked', all the parameters stored in the database will be displayed. This option is 'un-checked' by default. In 'un-checked' state, the parameters in the parameter catalog would be automatically filtered according to the Connector ID selected in the ID catalog. User can change the selection in the Connector ID table and the parameters under the selected ID will be displayed. For example, in the Figure 25, the Connector ID number 8 (name: CD outer diameter) is selected by the user. In the parameter table, only the parameters, which represent this physical quantity, are displayed. These parameters are from different modules and have

different labels but they represent the same quantity *outer diameter of a clutch disc (CD).*

- **User Controls**

On the left side of both catalogs (*DataGridView* elements), there are a few user control elements. These custom user controls are pre-defined and are used frequently in CluSys; therefore, they are used here as well for a seamless user experience. The *plus* button adds a row in the table. The *cross* button deletes the selected row. The *up and down* arrow buttons are disabled in the code, but could be implemented later if needed.

- **Saving changes to the database**

To save changes to the database, there is a dedicated button in the bottom left corner of the dialog. This button triggers the '*validate and save'* mechanism implemented in the application, which will be explained later under the functions. If the data entered is valid, it is saved to the database. In case of invalid or empty entries in the tables, an error message is shown to the user, highlighting the invalid rows in the dialog. The *OK* button in the dialog triggers the same *validate and save* mechanism and closes the dialog if everything is saved successfully. The *Cancel* button closes the dialog discarding any unsaved changes.

7.1.2 Parameter Dictionary Class Diagram

Appendix A shows the UML (Unified Modelling Language) class diagram of the parameter dictionary application. Each sub-project in CluSys has the same base class structure. The sub project *Intercommunication_DB* inherits the same structure and communication between different classes with a few new ones. Apart from *Mediator*, *Settings*, *Object* and *Dialog* classes, which are already introduced in the previous chapter, *DatabaseHandler* class is also inherited from a base class. Other than that, there is an *IntercomDB_DataStructures* class, which is not inherited from any base class. In the diagram, only few of the members are displayed. The communication between the classes is also shown and the member functions will be explained one by one in the following sections. Only important functions are explained in detail. First, *IntercommDB_DataStructures* class (*Data Structures for Intercommunication Database*) will be described.

7.1.3 Data Structures Class

The *Intercommunication Database* application required two basic data structures for defining Connector ID and parameter. The same data structures are used by all the classes to define vectors for the lists to exchange data. The complex structure of the CluSys classes and division between the *Object* and *Dialog* hindered an easy exchange of the data. This class defines the basic data structures without any methods, just to enhance the interoperability between different programming languages. Following is a code snippet showing the data structures *Connector_ID* and *Param.*

```
enum DBStatus {ADDED, MODIFIED, UNMODIFIED, DELETED};

struct Connector_ID {
    int             _ID;
    std::string     _Name;
    std::string     _Quantity;
    DBStatus        _Status;
};

struct Param {
    int             _Index;
    std::string     _Label;
    std::string     _Module;
    std::string     _Type;
    int             _ID_x;
    int             _ID_y;
    std::string     _Expert;
    DBStatus        _Status;
};
```

Figure 26: Basic data structures defined in DataStructures Class

The above figure (Figure 26) shows the enumeration type `DBStatus` which is used to define the status of the added entry. This status is changed when the *Connector Id* or *Parameter* is *modified*, so that the changed data is updated in the database without creating duplicate entries with the same data. This `DBStatus` element is very important for concurrency during the data exchange. By accessing this class, The Object and Settings class can use the same data structure. In the future any change in the data type could be easily done, just by changing this class. For example, if the parameter label, which is a string type, needed to be implemented as an integer type, this could be achieved by changing the type in this class. Figure 27 shows the *IntercommDB_DataStructures* class.

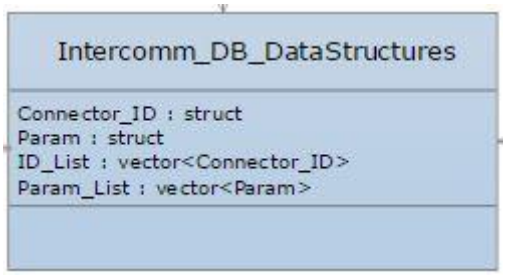


Figure 27: IntercommDB_DataStructures class

7.1.4 Database Handler Class

The intercommunication database handler class *IntercommDB_Handler* is derived from the base class *DatabaseHandlerBase*. This class is used to connect and exchanging data with databases. Figure 28 shows the *IntercommDB_Handler* class:

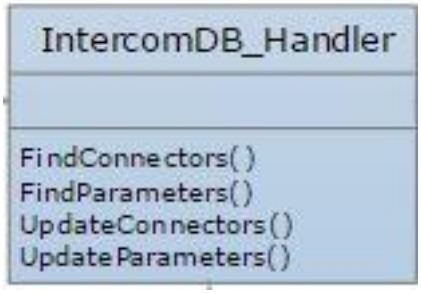


Figure 28: IntercomDB_Handler class

In *DatabaseHandlerBase* class, the basic functionality of the database handler is implemented. For example, the implementation of the database can be seen in the following code snippet:

```
IntercommunicationDataBaseImplementation()
              {
                     _Driver = gcnew DBI::MySQLDataBaseDriver;
                     _DataTable = gcnew System::Data::DataTable;
                     _Spec = gcnew DBI::Specification;
              }
_pImpl = new IntercommunicationDataBaseImplementation();
```

Instances of `_Driver` are used to connect and place queries to the database. `_DataTable` is the relation in a database. `_Spec` is the specification of the database. It covers different aspects of storing and retrieving the data in the database. There are two main methods to specify the retrieval of data from the database.

`_Spec->Condition`: The condition specifies which rows are filtered from the database. For example:

```
spec->AddCondition(gcnew DBI::Condition(gcnew System::String("ID"),
DBI::Condition::Operator::Equal, gcnew DBI::IntValue(42)));
```

The above code *adds* a condition to the specification, the condition operator is *equal to* and the condition operand is *ID*. This condition retrieves only the rows that have the *ID* equal to "42".

`_Spec->Selection`: The selection method specifies the retrieval of data with respect to the columns. For example:

```
spec->AddSelection(gcnew DBI::Selection("Label"));
```

The above line specifies selection criteria to retrieve the data from the database. Only the column "Label" will be retrieved from the database. If no *condition* or *selection* is specified in the specification, than all the data is retrieved.

The main member functions of the class are as follows:

- **FindConnectors()**

This is the function to get the *Connector ID* table from the database. The function first creates a `_DataTable` and then fills the table with the data from the database based on the following specification:

```
System::Data::DataTable^ table    = _pImpl->GetDataTable();
DBI::Specification^ spec          = _pImpl->GetSpecification();
spec->AddSource(gcnew DBI::Source("Connector"));
bool success = _pImpl->GetDriver()->Find(spec, table);
```

In the above code snippet, table and spec are created. A source "*Connector*" is added to the spec. The last line of the snippet calls the `GetDriver()->Find` function of the `_pImpl`, which is the instance of the implementation of the database. `Find()` is passed the specification and the table, in which the data is to be filled according to the specification.

- **FindParameters()**

This function retrieves the parameter table from the database. The function is similar to the above-mentioned function, except it adds the "*Parameter*" as the *source* instead of the *Connector*.

```
spec->AddSource(gcnew DBI::Source("Parameter"));
```

Both of the `Find()` functions look for the specified table and if found get the data and fill the `vector<>` of data types accessed from the *IntercomDB_DataStructures* class.

- **UpdateConnectors()**

This function takes the Connectors vector and updates it to the database based on their `DBStatus`. If an element in the vector has the *ADDED* status, it is added as it is at the end of the table.

```
table->Rows->Add(row);
```

If it is *MODIFIED*, the corresponding Connector ID in database is updated. If the `DBStatus` is *UNMODIFIED*, nothing is done. If the entry has the `DBStatus` *DELETED*, then the corresponding entry in the database is deleted as well. During updating the entries in the database, each updates entry is '*reset'* with `DBStatus` *UNMODIFIED*.

- **UpdateParameters()**

This function works in the same manner as the above function. It gets the vector and checks each element's `DBStatus` value and update according to that. As the

```
_invalidparams.push_back(parameters[i]._Index);
```

ID, (x-axis) value (_ID_x) and *ID (y-axis)* value (_ID_y) are the foreign keys for this table. They are updated only if the value is already defined in the Connector IDs table. In case of a non-existent value for the foreign key columns, an error string is returned. Along with the error message, the index of the parameter at which the error is occurred is also saved in a vector `_invalidparams`.

7.1.5 Database Object Class

The *Object* class of the database application (parameter dictionary) is the main class where all the program logic is implemented. This class has a lot of member variables and member functions. A compact view of the class can be seen in Figure 29.

Figure 29: Intercommunication Database Object class

The main methods of the *IntcomDB_Obj* class are:

- **ReadDatafromDB()**

  At initialization of the application, first an instance of database *handler* is created. *Handler* initializes the database and creates a connection with the database by passing the database *identifier*, *port number*, database name, etc. There is no

```
handler = new Database::IntercommunicationDataBaseHandler();
handler->Initialize("de020353;Port=9013", "intercommunication",
"Clusys", "Clusys");
ReadDataFromDataBase();
```

  access authorization or restriction mechanism, so that all the *Experts* could access and make changes to the database. Once the connection is established, `ReadDatafromdataBase()` function is called. In this function, following handler functions are called:

```
handler->FindConnectors(ID_LIST);
handler->FindParameters(_PARAM_LIST);
```

These functions *fill* the ID_List and Param_List vectors.

- **UpdateDataToDatabase()**

This function writes back the data to the database using the database handler:

```
handler->UpdateConnectors(ID_LIST);
handler->UpdateParameters(_PARAM_LIST);
```

If the data is successfully written back to the database, this function returns true, in case of an error, an error message is prepared based on the database error string. This function analyses the contents of the database error string to find specific information, so that the user could be informed about the type of the error.

All the remaining functions of the *IntcomDB_Obj* are given in the appendix A.

7.1.6 Database Mediator

The mediator class of intercommunication database *IntercomDB_Mediator* communicates between the dialog and the object classes.

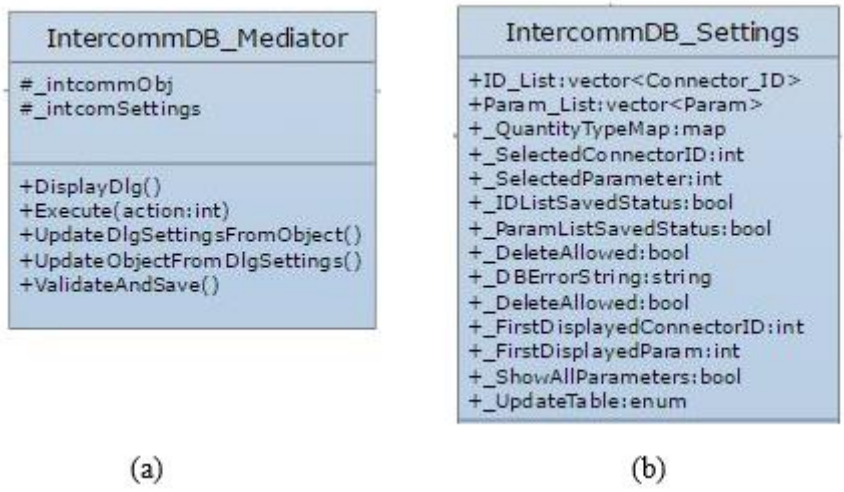


Figure 30: (a) IntercomDB_Mediator class. (b)IntercommDB_Settings class

This class has the protected objects `_intcommObj` and `_intcommSettings` of *IntercomDB_Obj* class and *IntercomDB_Settings* class respectively. The `DisplayDlg()` function displays the displays the dialog by passing the respective dialog ID. The `Execute()` is the main function of mediator and it is called by the special *callback* function of the settings. It executes some *action* based on the action enumerator set in *Settings* with the help of switch cases. Each case calls the

corresponding functions in *Object* class. The `UpdateDlgSettingsFromObj()` function updates the *Settings* from *Object* as seen in the following code snippet:

```
{
	_intcomSettings->_IDList.clear();
	_intcomSettings->_ParamList.clear();
	_intcomSettings->_IDList = _intcomObj->get_ObjectIDList(_intcomObj-
>_ConnectorIdListSearchTerm);
	_intcomSettings->_ParamList = _intcomObj->_getParamList(_intcomObj-
>_ParamListSearchTerm);
	return(true);
}
```

First, both vectors are cleared and then the updated lists from *Object* is fetched passing the search term entered by the user in the search bar. The fetched list would be with respect to the search term and the filter selected.

The `UpdateObjFromDlgSettings()` function does the same in reverse order and simply updates the *Object* `_IDList` and `_ParamList` from the *Settings*.

7.1.7 Database Settings

The settings class is derived from *CluSysBaseSettings* class. This class has the *enumerator* type, which defines different *actions* that are being triggered when an event occurs in the dialog. The callback function calls the *Execute* of `IntercomDB_Mediator` with a value of these actions. This class has only member variables for data exchange between the *Object* and *Dialog* classes. No function is implemented in this class as can be seen in the Figure 30 (b). For example, `ID_List` and `Param_List` vectors are for the catalogs to show on the dialog. The `_QuantityTypeMap` is a *map* type that is used in CluSys to access the available quantity types for the parameters.

7.1.8 Database Dialog

IntercomDB_Dlg class is the main application GUI class. It has a lot of GUI elements and events, most of which are created automatically using the interactive *Designer* of Visual Studio. In the Figure 31 some of the functions and attributes can be seen. The attribute `_pSettings` is the object of *IntercommDB_Settings* class. In *Dialog* class, no data is exchanged directly; all the exchange is done through *Mediator* using this `_pSettings`.

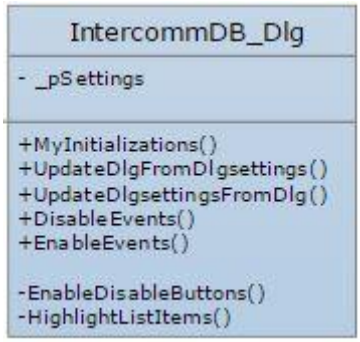


Figure 31: IntercommDB_Dlg class

The `MyInitializations()` function is derived from the *BASECLASS* `MyInitializations()` and it initializes the dialog with default settings. `UpdateDlgFromDlgSettings`() function updates the dialog with respect to the *_pSettings*. `UpdateDlgSettingsFromDlg()` function writes back all the changes done by the user using the user interface to the `_pSettings`. These changes are then written back to the *Object* class members. `DisableEvents()` and `EnableEvents()` are also default functions derived from the *BASECLASS*. These functions enable or disable the events during data read or write process. This mechanism makes sure that the data on the dialog or the data in the object is same and is not changed during the exchange process. The private functions `EnableDisableButtons()` is called with every event ion the dialog to enable or disable appropriate buttons to make user experience better. The `HighlightListItems()` function is called to highlight respective rows or columns to assist the user. For example, is case of an invalid ID or a duplicate *Label* of a parameter, it would be highlighted as red. A new row or a row that is being edited is highlighted as pale yellow (see Figure 32).

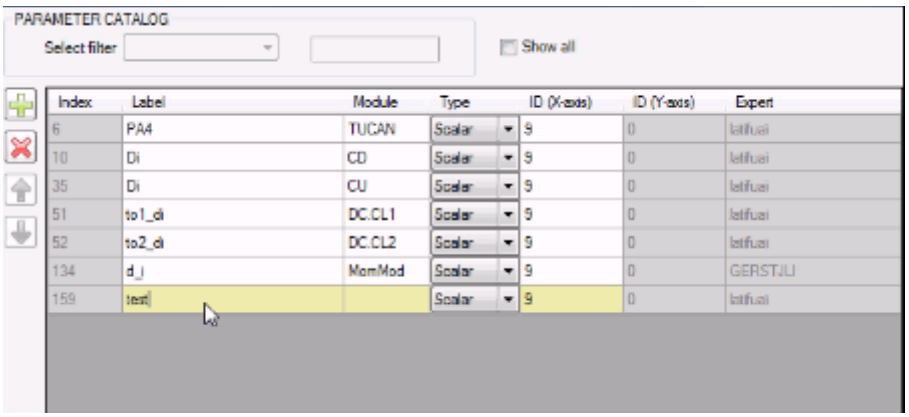


Figure 32: A highlighted parameter while editing

7.2 Automatic Intercommunication Interface

As explained in the chapter 6, the automatic intercommunication interface accesses the parameter dictionary and based on that recognizes the parameters in the simulation modules. To integrate the interface in the CluSys platform, it is placed in the *Container_Obj* class.

7.2.1 Container Approach

In CluSys development, the concept of *container* is used to guarantee the abstract and flexible approach for simulation experts. A *container* can contain a single simulation model, a collection of simulation models or a collection of these collections and so on. The program treats all the containers similarly and calculates and run the simulation in a defined order for each of the containers. This approach makes it easier to work in CluSys on two different abstraction levels. In *Expert-mode*, this container is *open* and experts can define anything in the container, the simulation models, GUI for the models, intercommunication between different simulation models, etc. For end user the container is *closed* and it can be seen like a black box.

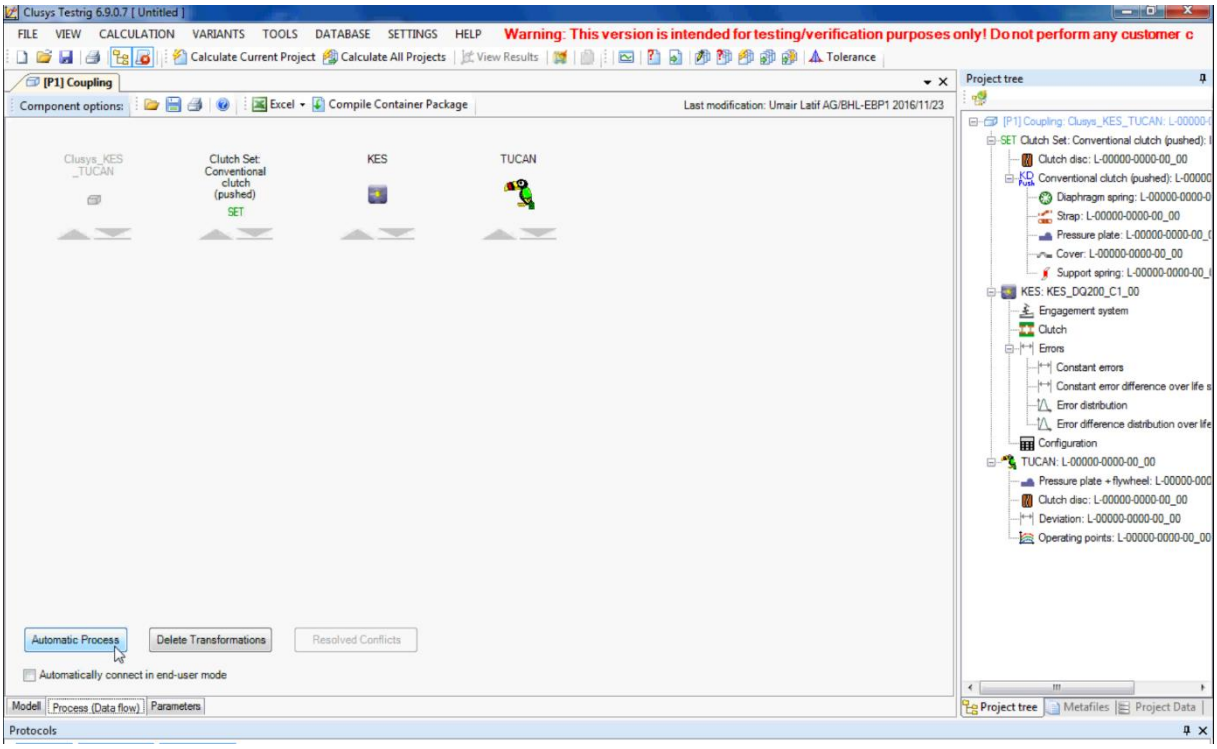


Figure 33: Window showing the Data Process tab page

7.2.2 Container Process Window

In Figure 33 a snapshot of a user interface can be seen where an expert user can see the project tree on the right hand side with different simulation modules and their components. On the left side of the window, the modules are visible without any intercommunication defined between them. In the bottom left corner, there are three buttons and a check box. This window is only visible to the expert users.

- **Automatic Process Button**

This button activates the automatic intercommunication interface process and creates the intercommunications between the modules in the project automatically. The experts do not have to define the data process using the manual user interface.

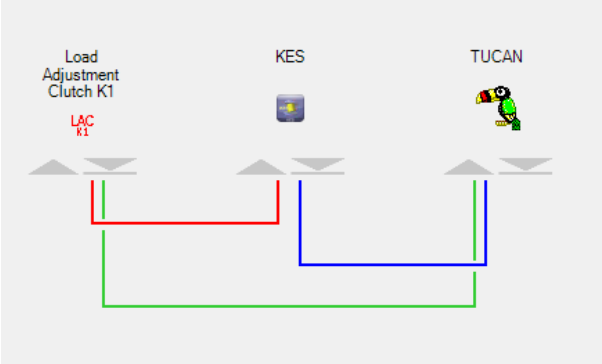


Figure 34: Intercommunication between different modules shown as colored lines

- **Delete Transformations**

This button deletes the transformations or data exchange processes between the modules. This button is to facilitate the experts in case they want to experiment with different modules before they finalize their simulation package. The word *transformation is* used here because, in CluSys, the data process between two modules is known as a *data transformation.*

- **Resolved Conflicts Button**

This button is only *enabled* when there is a data conflict during automatic data process. This button opens the *Conflict resolution* dialog.

- **Automatically Connect in End-User Mode Check-box**

There is a *checkbox* under these buttons, which enables the automatic intercommunication interface for the end user. As the end user cannot see this window, experts has to enable that option if they want that their simulation

package or *container* is automatically interconnected every time a user opens it or replaces any module with another one.

7.2.3 Container Object Class

The manual data processing is implemented in the *Container_Obj* class, where a dialog is opened whenever an expert wants to define the process for data exchange between modules. For that reason, the automatic interface is implemented in the *Container_Obj* class as well. This will achieve the seamless integration requirement and allow to use the already developed functions without rewriting the basic functions for data processes.

The basic program logic behind the container approach is implemented in the *Container_Obj* class. This class has a plethora of member functions, which are out of the scope of this document and will not be explained here. The functions and attributes that are created to implement the automatic intercommunication interface can be seen in the following figure (Figure 35) and only the important functions will be explained here in detail.

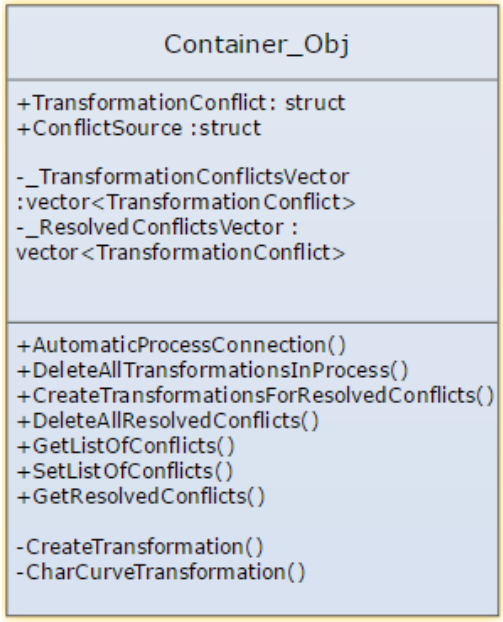


Figure 35: Container_Obj class

The data structures `TransformationConflict` is created for storing the data conflicts during automatic intercommunication process. Following snippet shows the data structures:

```
//struct for storing the conflicts
enum RESOLVED_CONFLICT_STATUS{ADDED, MODIFIED, UNMODIFIED };
struct ConflictSource
{
    std::string                 _Label;
    std::string                 _ReferenceLabel;
    std::string                 _Denomination;
    bool                        _IsSelected;
};

struct TransformationConflict
{
    int                         _Index;
    int                         _ConflictType;              /
    bool                        _IsResolved;
    std::string                 _DestinationModuleLabel;    /
    std::string                 _outputLabel;
    std::string                 _ReferenceLabel;
    std::string                 _OutputDenomination;
    std::vector<ConflictSource> _PossibleInputs;
    RESOLVED_CONFLICT_STATUS    _ResolvedConflictStatus;    /
};
```

Figure 36: Data structures for conflicts

The main member function that implements the automatic intercommunication functionality is `AutomaticProcessConnection()`.

7.2.4 Automatic Intercommunication Logic

The concept of automatic intercommunication logic is based on accessing the parameter dictionary and the simulation modules. The simulation projects are created using the container approach and therefore the automatic intercommunication logic is also abstract and treats the containers same without any knowledge of the actual simulation model.

- **Connecting to the database**

  Following code creates a connection with the database and gets a list of saved parameters.

```
handler = new Database::IntercommunicationDataBaseHandler();
handler->Initialize("de020353;Port=9013", "intercommunication",
"Clusys", "Clusys");
handler->FindParameters(_IntcomDBList);
```

- **Recognizing parameters**

  Once the interface has the parameters, it can now access the container object. First it gets the number of *children* (that is, the models in a container) in the container

```
auto SourceList1 =
ContainerProcess_Helper::CreateProcessInputInfoList(child1);
```

by `GetVPartInfo().size()`. Then it goes through all the children one by one to get the input and output parameters lists.

`CreateProcessInputInfoList()` will get the list of input parameters for the said child (*child1*) of the *container* and `CreateProcessOutputInfoList()` gets the output list. Then it iterates through all the members in the list and check if any member has the same *label* as in the database list. To check that `find()` function is used, which search for the same sequence of a characters in a string.

```
if (Referencelabel.find(itDB._Module + "." + itDB._Label) !=
std::string::npos)
```

The `Referencelabel` is the label of a parameter in a simulation model, whereas the `itDB._Module + "." + itDB._Label` creates a label based on the information from the database combining label and the module name. This has been already discussed in detail in the section 5.3.2. If both labels are same, the parameter is saved in a new vector `_FoundParametersInSourceList`. This vector also saves the Connector ID of a parameter from the database list.

```
auto bc = this->GetCharCurveFromLabelPath(this, label);
if (bc != NULL)
if (bc->GetDataSourceType() != GFCore::DataSourceType::PROCESS_IMPORT)
       {
       _ParamInfo._Type= IntercommDB_DataStructures::ParamType::CURVE;
       _ParamInfo._Id_x     = itDB._ID_x;
       _ParamInfo._Id_y     = itDB._ID_y;
       _ParamInfo._Label    = iter._Label;
       _ParamInfo._ReferenceLabel = iter._ReferenceLabel;
       _FoundParametersInSourceList.push_back(_ParamInfo);
       }
```

In this way, the interface has enough information about a parameter to connect it logically to another one based on the underlying physical quantity of the parameter. To prevent a *false conflict* (see section 5.2), if a target parameter from an output list is already connected to a source parameter, it is skipped and not saved in the `FoundParametersInDestList` vector. This is checked by the helper function `GetDataSourceType()`.

- **Defining Data Process**

Once the interface has the parameters from destination or *target* and *source* modules, it goes through the *target* parameters list and checks one by one if any of the *source* has the same ID (that means the same physical meaning). Following snippet shows a `copy_if()` function of the C++ *Standard Library* [23]. As a

```
std::copy_if(_FoundParametersInSourceList.begin(),
_FoundParametersInSourceList.end(),
std::back_inserter(_MatchedSourceParams),
[&](IntercommDB_DataStructures::ParameterInfoForInterface p)
{return (id == p._Id_x) ; });
```

fourth argument to this function, a *lambda function*[5] is used. The above code copies the parameters with same ID to the `_MatchedSourceParams` vector. Depending on the *target* parameter type, only the *source* parameters with the same type are compared for a *match*. If there is only one *matched* parameter against a *target* parameter, then an appropriate `CreateTransformation()` function is called.

The `CreateTransformation()` function defines a data exchange process between a scalar *target* and *source* parameter.

```
CreateTransformation(label_modul_1, _MatchedSourceParams.at(0)._Label,
DestParam._Label)
```

- **Defining Data Process for Characteristic Curves**

If the parameter type is characteristic curve, another function `CharCurveTransformation()` is called for defining the data process between two curves.

```
CharCurveTransformation(label_modul_1,DestParam, _CurveParameters)
```

This function takes the *target* curve (`DestParam`) and the all the curves in the *source* list if at least one axis of that curve is same as the *target* curve's axis. This function is more complex and first, it checks for a possibility of a *direct curve*

[5] Lambda function is a C++ language facility that is provided since C++11 specification to allow programmers to define anonymous functions "*in-place"* or right where they are needed [24] [25].

transformation by checking if both *target* and *source* curves have exactly the same dimensions, which means they have same physical quantities representing the x and y-axis. After that, the function searches for possible *combination curve transformations*. It checks if two curves in the *source* list have one axis equal to the *target* curve and the other common between the two of them (for details see section 6.2). If such a *combination* exists, then it is also saved in the `_PossibleCurveTransformations` vector. After all the *source* curve parameters are checked against a *target*, it is checked if the vector for possible curve transformations has more than one elements. If it has more than elements, it is considered as a data conflict and no transformation is made. If it has only one possibility, the transformation is created in the end of the function.

7.3 Data Conflicts

As discussed in the section 6.3, in a chain simulation process, there is a possibility of data conflicts. In this implementation, the controlling parameter is the *target* parameter, so there is only one type of data conflict that could arise, that is multiple *source* parameters available for a single *target.* During the previous step of defining data transformations, if there are more than one *source* parameters in `_MatchedSourceParams` vector, this is saved as a data conflict in a separate vector `_TransformationConflictsVector`. Figure 36 shows the data structure used for storing the transformation conflicts. These stored conflicts are shown to the user for resolving through the dialog *Conflict Resolution*.

7.3.1 Conflict Resolution Dialog

The conflict resolution dialog (see Figure 37) can be opened by the experts by clicking on the *"Resolved Conflicts"* button in the *Container process* window. The button is only enabled if there are any conflicts during the automatic intercommunication process. The number of conflicts detected and the number of conflicts already resolved automatically or by the user is also added to the label of the "Resolved Conflicts" button in the form "[2/5]" which means *two out of five total conflicts* are resolved. The conflict resolution dialog has two tab pages, one for showing *Current Conflicts* and the other for showing all the conflicts that are resolved and saved in a specific project. On *Current Conflicts* tab page, the right hand side table shows the *target parameters*, whereas the left hand side table shows the possible *source* parameters that are collected by the conflict management mechanism.

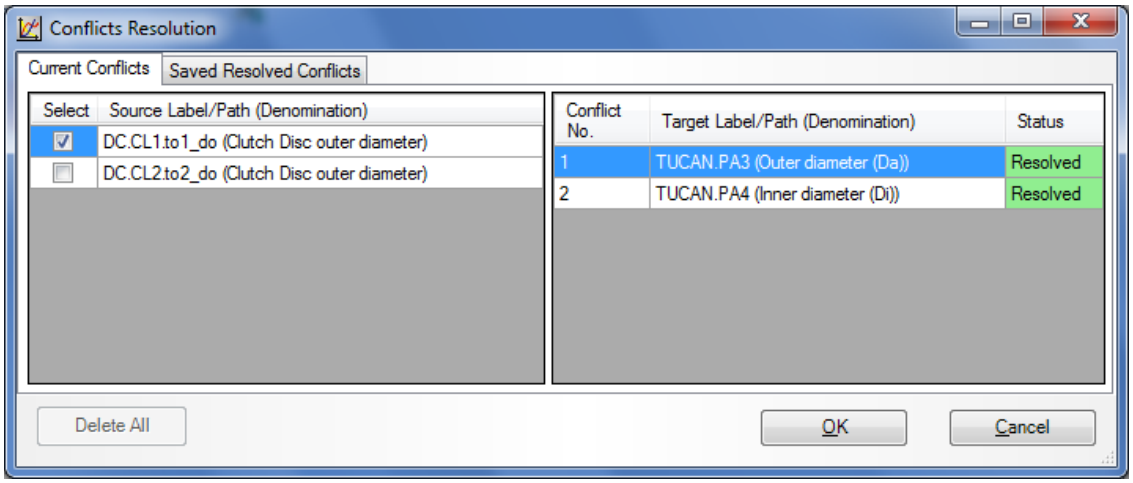


Figure 37: Conflict Resolution dialog

When a target is selected in the right table, corresponding source parameters are displayed on the left. Each row displays the *label* and the *denomination* (a freely chosen phrase to describe the parameter) of the possible source parameter. There is a *Select* column displaying a checkbox in each row. This checkbox can be used to select a *source* for a target and the target *Status* will be set to "*Resolved*" and the background of the *Status* cell will be set to *green*.

The second tab page "Saved Conflicts" displays all the conflicts saved that are resolved by the user or automatically by the interface. The saved conflicts list could become larger than the actual conflicts for a particular combination of modules. For example, if an expert uses one module in a project resolves its conflicts and then replaces it with another one. The conflicts will be updated according to the new module and the expert has to resolve them as well. All these and previously resolved conflicts will be saved with the project data, so that if the user decides to replace the module again with a first one, the conflicts will be automatically resolved by the interface using these "*saved conflicts*".

7.3.2 Conflict Resolution Logic

Figure 38 shows the class diagram of the conflict resolution dialog. The `Object` class `Container_Obj` is accessed by the `ConflictResolution_Mediator` class, because the same class object collects the *Transformation Conflicts* in `_TransformationConflictsVector`.

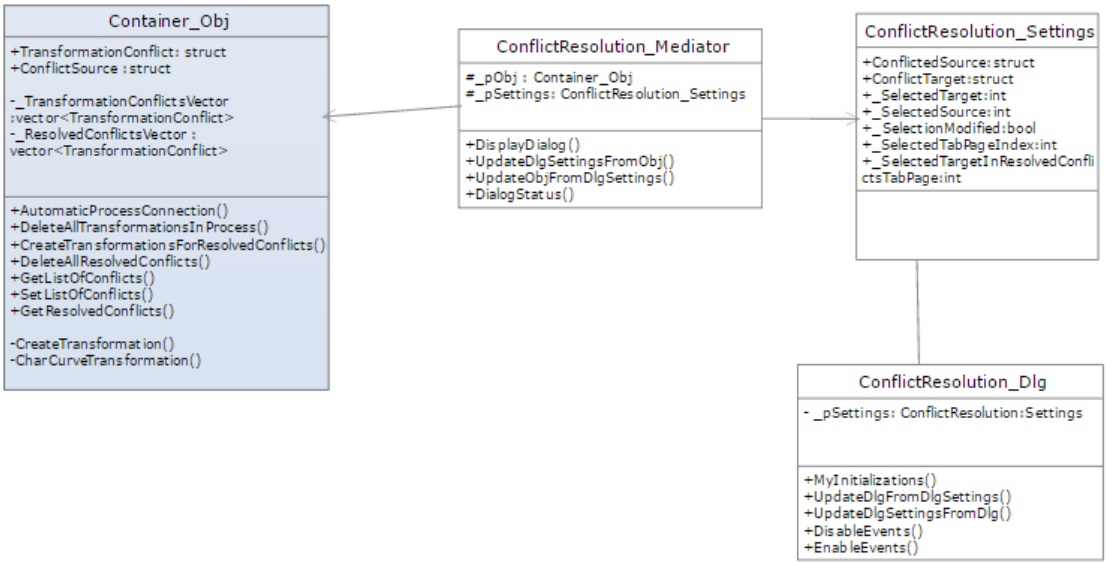


Figure 38: UML class diagram of Conflict Resolution

The `ConflictResolution_Mediator` class accesses the `ConflictResolution_Settings` by creating an object of the class `_Settings`.

```
_Settings = new ConflictsResolution_Settings();
_Settings->_DlgId = GUI::DlgID::CONFLICTS_RESOLUTION_DLG;
```

The `UpdateDlgSettingsFromObj()` function updates the `_Settings` vectors with the conflicts collected by the `_Obj`. The `UpdateObjFromDlgSettings()` writes back the user input from the `_Settings` to the `_Obj`. The function `DialogStatus()` returns the dialog status from the `_Settings->_DialogResult`. If the user pressed *OK* button after changing a selection in the dialog, the status message "true" as a *bool* type will be returned, the mediator calls the `CreateTransformationsForResolvedConflicts()` function of the `_Obj`.

- **Data Process For Resolved Conflicts**

  After the conflict resolution dialog is displayed with the collected conflicts, the user selects a *source* parameter for a specific *target* parameter.

  This decision is exchanged to the *_Obj* through the *Mediator* class. The *Mediator* calls the `CreateTransformationsForResolvedConflicts()`

function. This function defines data processes for the conflicted parameters based on the user decision. If a conflict is successfully resolved, the decision is saved in a `_ResolvedConflictsVector` vector. This vector is saved with the project data in an XML file using the `SaveXML()` function of the `Container_Obj`. The structure of the saved XML data can be seen in the appendix B.

- **Automatic Conflict Resolution Using Resolved Conflicts**

When a project is loaded, the `_ResolvedConflictsVector` vector is loaded as well using `LoadXML()` function. If the "*Automatically connect in end user mode*" option is *checked* in the process dialog by the expert, the `_AutomaticProcessConnection()` function is called and if it has found any conflicts, it will call the `CreateTransformationsForResolvedConflicts()` function. This function will check if there is any resolved conflict. If there are resolved conflicts present, then the function will check one by one, if any of them is same as the current conflict. If there is a same conflict, the saved decision will be chosen and a transformation is created without prompting the user.

Summary

In the implementation phase, first the Intercommunication Database or the *parameter dictionary* is implemented following the concept proposed in chapter 6. The implementation required that the class structure of CluSys should be taken into account and therefore all the classes of intercommunication database are derived from the *BASECLASSES*.

A database application is developed for creating and maintaining the database. The application has a simple user interface, which is developed using the *OK-Cancel* template available in CluSys. The application checks the validity of data before it is saved to the database. The database design implements the concurrency and integrity of data.

In the second part of implementation phase, the automatic intercommunication interface is developed. The interface is placed parallel to the already present mechanism of manual data exchange process. The interface is implemented within the *Container sub-project*, which makes it abstract on a broader level. The interface is based on the algorithm provided in chapter 5. It accesses the parameter catalog from the database, which contain the parameters and their physical meanings in the form of unique Connector IDs. It also accesses the input and output

parameters of the modules and follows the algorithm to find parameters with same physical meaning and defined data processes between them. In the end, the *Conflict Resolution* mechanism is implemented in case there are multiple intercommunication possibilities between modules. These possible intercommunication options are collected and the user is prompted to solve them individually with the help of a simple *Conflict Resolution* dialog. These user decisions are saved with the project *XML file* and are automatically implemented if the same conflict arises later. The implementation covers all the requirements presented in chapter 4 successfully.

Chapter 8

8. Tests and Results

The implemented Intercommunication Interface along with the *parameter dictionary* provides a mechanism to define data exchange between the simulation modules automatically. The system has been tested with different module combinations of variable complexity. To test the results of the implemented functionality, some dummy projects were created with different simulation modules used by the product engineers. In the end, the results derived from the test projects are summarized.

8.1 Test Projects

The combinations in these dummy projects were not the same as used in actual simulation projects. Their main purpose was to test whether the intercommunications defined by the interface are correct or not. To compare the correctness of the intercommunication or data exchange processes between the modules, some pre-defined and pre-compiled examples of simulation packages are used. The next section will discuss a couple of examples with their results.

8.1.1 Test Project 1

In the first test project, three simulation modules are used, a clutch system, which is called *Clutch Set*, a *KES* module and a *TUCAN* module, both of which are the same as discussed in chapter 2, section 2.3.2.

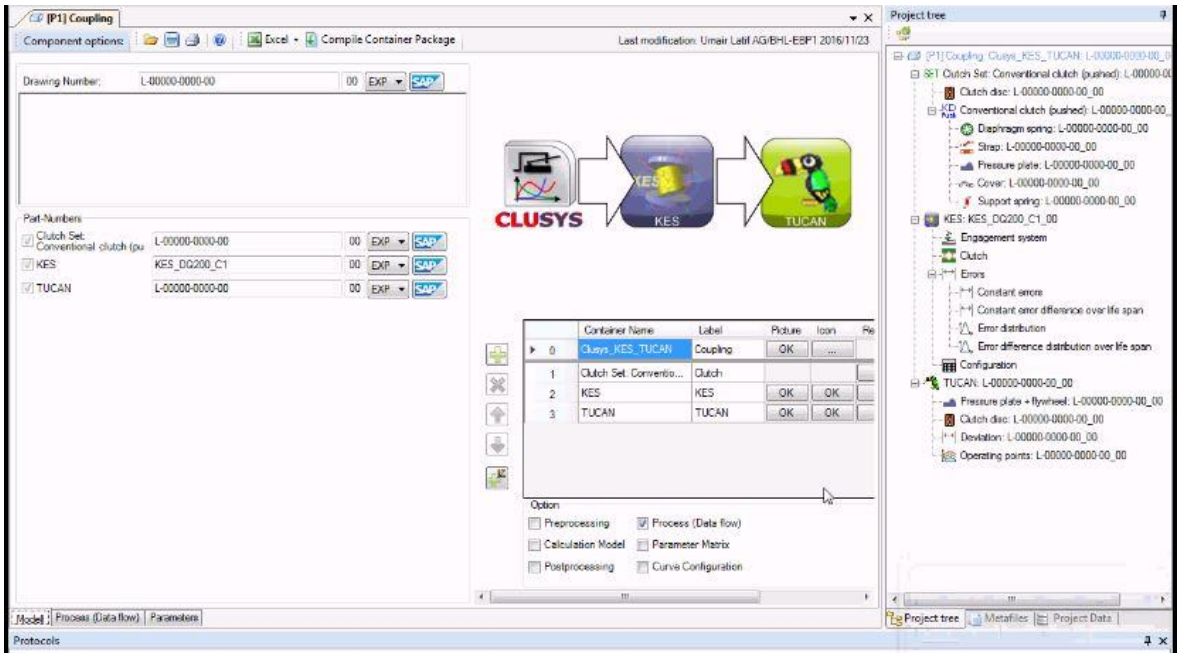


Figure 39: Window showing the test project with Clutch Set-KES-TUCAN

Figure 39 shows the test project window, with the project tree in the right corner. These three simulation packages have a *clutch* sub-module in common, that means the values of the parameters in *Clutch Set* should be transferred to next modules. The other two calculation kernels will calculate their respective analysis results based on these parameter values.

First, the parameters from these modules are defined in the *parameter dictionary*. Figure 40 shows the parameter dictionary application. In the *Connector ID* table, ID number 8 is selected which is the "*CD outer diameter"*. Under this ID, various parameters are defined in the right side table. For example, the parameter "*PA3"* from module *TUACN* and parameter with the label "*Do"* from module *CU*, which is a component of the *Clutch Set* module.

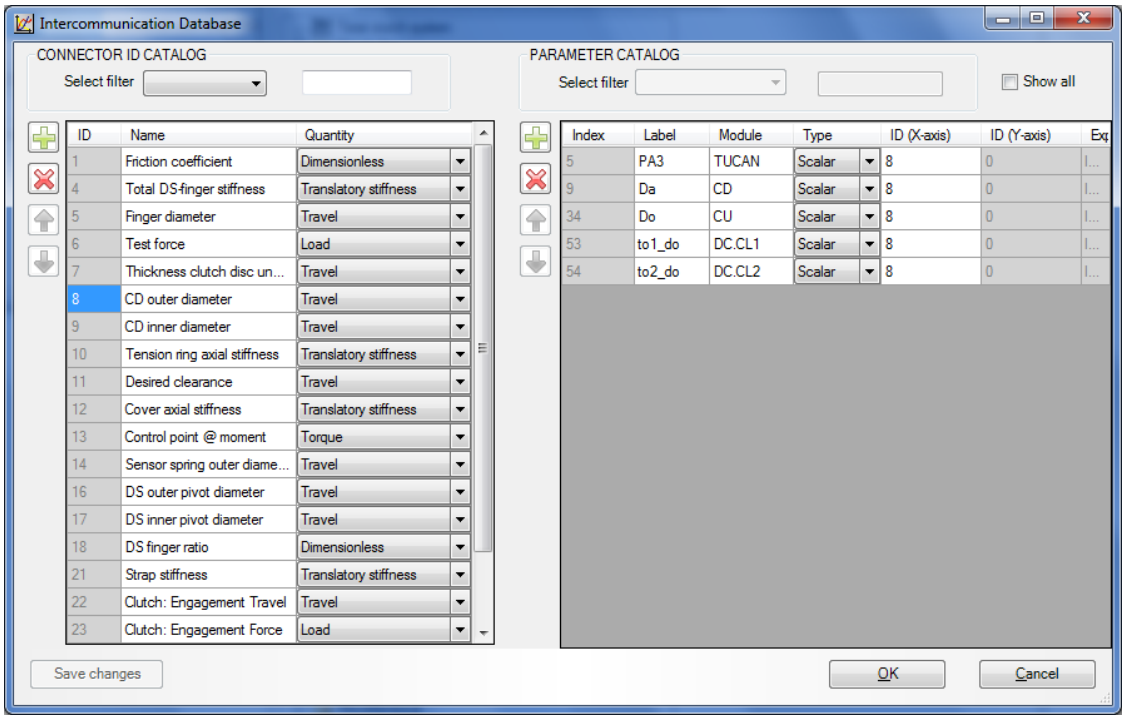


Figure 40: Parameter dictionary with parameters from test modules

The automation of this data exchange process is implemented in the interface. When the *Automatic Process* button was clicked in the *Data Process* tab page, the following message box was displayed.

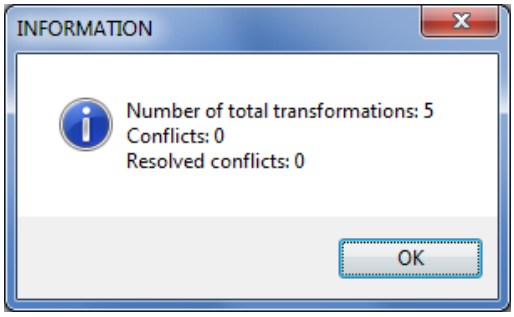


Figure 41: Message box with information about the automatic data processes

The message box (Figure 41) shows how many data exchange processes are defined automatically by the intercommunication interface. In this example, there are five automatically defined data exchange processes or *data transformations* created by the interface. Following figure (Figure 42) shows the *Data Assignment* window, which shows the defined data exchange processes.

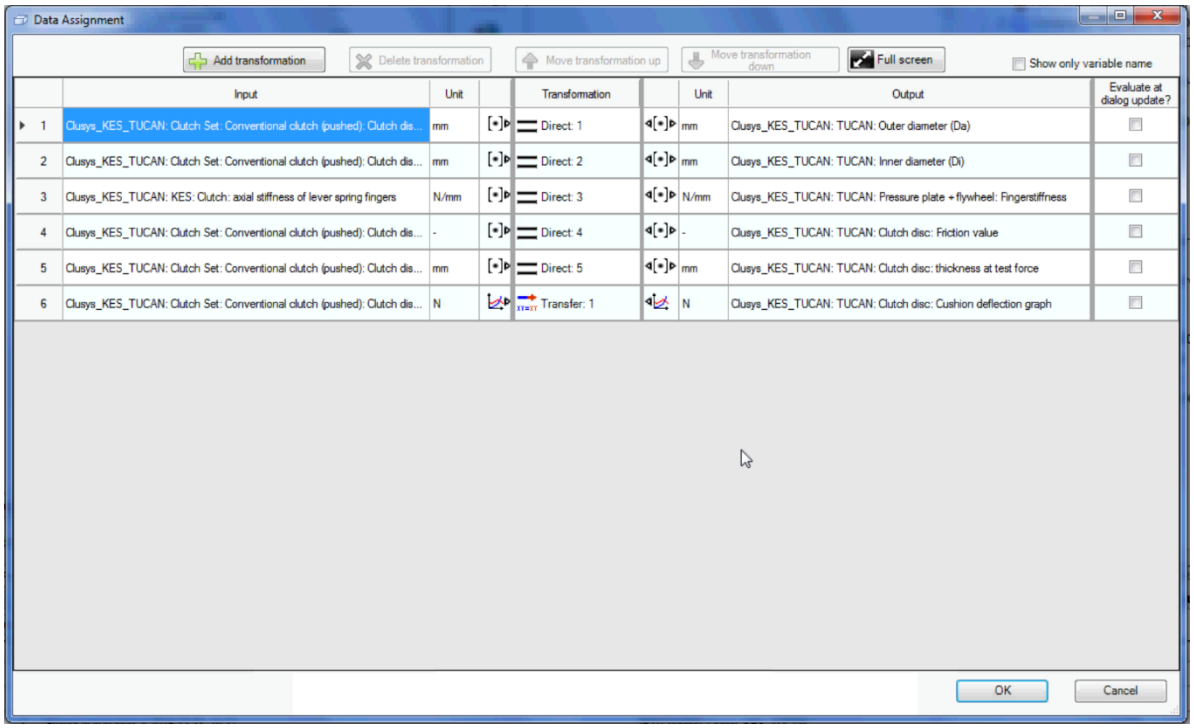


Figure 42: Data Assignment window showing automatically defined data processes

This window shows the data assignment processes defined between the parameters of *Clutch Set* module and the *TUCAN* module. The Table 1Table 4 below shows the *denominations* (full labels) of the parameters of both modules and the transformation type defined between them.

Input Parameter (Clutch Set module)	**Type of Data Transformation**	**Output Parameter (TUCAN module)**
Clutch disc outer diameter	Scalar direct	Outer diameter (Do)
Clutch disc inner diameter	Scalar direct	Inner diameter(Di)
Axial stiffness of lever spring fingers	Scalar direct	Finger stiffness
Clutch disc friction value	Scalar direct	Friction value
Clutch disc thickness	Scalar direct	Thickness at test force
Clutch disc cushion deflection	Characteristic curve direct	Cushion deflection graph

Table 4: Data assignment between parameters

The data exchange defined in the table above is correct and fully compatible with the physical meaning of the parameters involved. The first *assignment* is highlighted in the table, which shows the parameters that are mentioned before and can be seen in the parameter dictionary under the same Connector ID (see Figure 40). Each data transformation is defined between the parameters of same physical quantity, as defined by the user in the *parameter dictionary.*

8.1.2 Test Project 2

To test the *Conflict resolution* functionality, an example with a *Double Clutch Assembly* (German: Doppelkupplung or *DK*) is chosen. A DK module has two separate clutches in one assembly, which means that there are two sets of parameters with definitions of a clutch geometry.

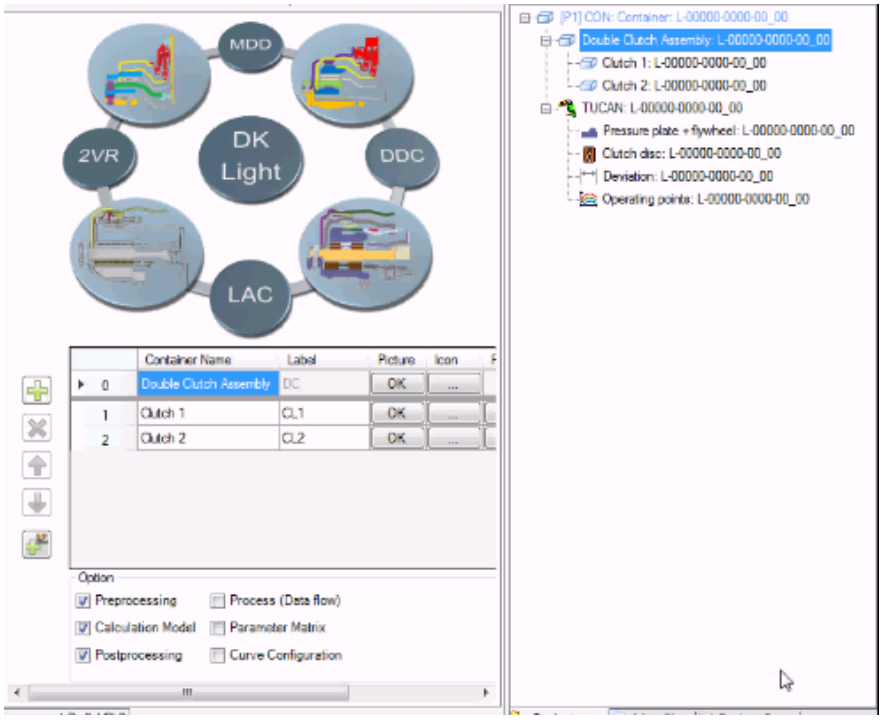


Figure 43: A Double clutch assembly (DK Light) module

In Figure 43, the *DK Light* module is combined with a *TUCAN* module. When the automatic process is run, the following message box is shown.

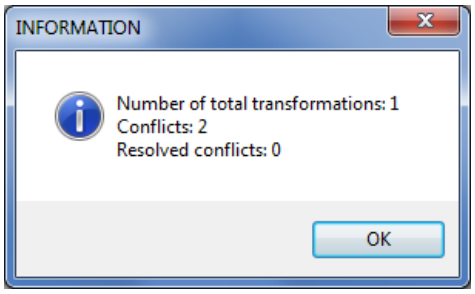


Figure 44: Message box in case of data conflicts

The message box (Figure 44) shows the number of conflicts that the automatic intercommunication interface has recognized.

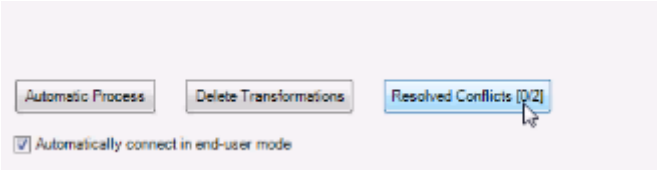


Figure 45: Resolved Conflicts button activated

The *Resolved Conflicts* button is now *enabled* and the *Conflict Resolution* dialog can be seen by clicking the button.

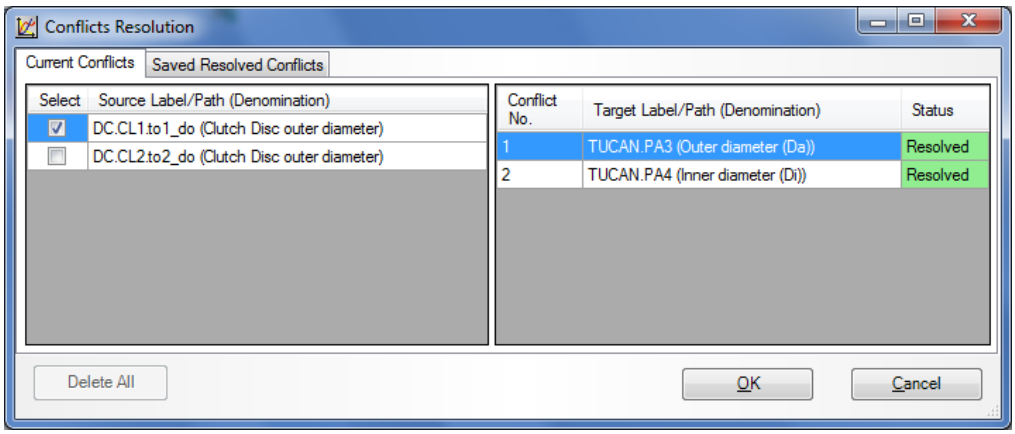


Figure 46: Conflict Resolution dialog showing the conflicts

The *Conflict Resolution Dialog* shows the target parameters and the possible source parameters for a selected target. In our test project, the *Double Clutch* module has two "Clutch disc outer diameter" for a target "Outer diameter (Da)" from *TUCAN* module. The outer and inner diameter of clutch disc from clutch 1 (CL1) is chosen in this case. The *Data Assignment* is done when the dialog is closed with *OK* button.

8.2 Results

The interface is able to create intercommunications between the parameters based on their physical quantity disregarding their name, module name or units. The only requirement for a correct data exchange definition is that the respective parameters are defined in the parameter dictionary. The experts have to define the parameters once in the database and then they could be recognized afterwards in any project. As was the case with the parameters defined for *TUCAN* module. They are detectable in both projects in our examples and were connected to different parameters of different clutch modules.

The conflict resolution mechanism implemented is also working seamlessly with the intercommunication interface. It collects the information about the data conflicts and prompts the user to resolve them manually by selecting the correct parameter. The resolved conflicts are saved in the *XML* file with project data.

Chapter 9

9. Summary and Outlook

This chapter states a summary of the whole work accomplished during this thesis. In the second part of the chapter, an outlook is provided with suggestions for future development and improvements.

9.1 Summary

The simulation platform CluSys used in LuK GmbH & Co. KG is also one of such simulation environments that provide a possibility of creating multidisciplinary chain simulation processes in which the individual simulation modules (or calculation kernels) created in different simulation modelling software could intercommunicate and exchange data to simulate an overall system. CluSys serve as the control program for these calculation kernels and take care of the input, running the simulation based on the user input and representation of the simulation results with respect to their formats. The intercommunication between the simulation modules was defined manually by simulation experts. The experts used a GUI to interconnect parameters based on their own knowledge about the modules. There existed a chance of error because of the ambiguous and non-standard naming convention. Apart from that, a simulation model normally has hundreds of parameters, interconnecting them not only required a lot of time but the manual mechanism was prone to parametrization errors. The manual approach hindered the use of *module replacement* mechanism, which allowed users to choose freely between multiple modules for a chain simulation to compare the results with different combinations.

The aim of this master thesis was to conceptualize and develop a solution to automate the data exchange process between the parameters of different simulation models in a chain simulation process. During the literature research phase, different approaches for defining data exchange between modules were analyzed. A few of them used a *configuration file* approach, which allowed user to define the data exchange between parameters in these files. This approach could automate the manual data assignment process for a defined project, but could not help if a single module is *replaced* in the same project. Another approach was to create an abstract mechanism for data exchange between the parameters by creating an interface. The interface could use a database of parameters, just like the configuration file

but more generalize and containing the information about the parameters based on their physical meaning, not just their names or the module names.

After the analysis of the researched approaches, the interface approach was selected as it is much more abstract and flexible and provides good compatibility with the already implemented solution in CluSys. For implementation of the interface approach, a database was required, which catalogs the physical meaning behind the parameters of simulation models. This database served as a *parameter dictionary* maintained by simulation experts. The database was placed on the LuK server so it is accessible worldwide in the LuK network. Currently there is no restriction or user authorization is in place to access the database but this could be changed in future. The database has two tables, which are inter-related to each other. One table contains the physical quantities, which have a unique *Connector ID*. The other table contains the parameter catalog, in which each parameter is defined in terms of their physical quantity using the *Connector IDs* from first table. Together these two tables created the parameter dictionary. The GUI for the parameter dictionary was developed using .NET technology and C++/CLI. The structure of the application as well as the implemented interface is compliant to the overall CluSys structure.

The automatic intercommunication interface was implemented in the *Container* sub-project of CluSys. The *Container* approach makes the interface more flexible in its functionality as the implementation is independent of the simulation modules. It recognizes the parameters with respect to their physical meaning, disregarding their label, module name or unit. The interface then automatically interconnects parameters of different modules based on their physical meaning, so that the value of the *source* parameter is automatically transferred to the *target* parameter if they represent the same physical quantity, performing appropriate *data transformation*, if needed. If a user *replaces* a module with another one, the intercommunication and data exchange processes will be lost between the parameters. The interface connects the parameters again automatically. So for an end user, it would be same as opening a different pre-compiled simulation package.

In case there are more than one *source* parameters available for a *target* parameter, the interface recognizes this as a data conflict and prompts the expert user to resolve the conflict. The user can select one input for a particular *target*; this user decision is stored as a *Resolved Conflict* with the project data as an *XML* file.

9.2 Outlook

With this implementation, an abstract and flexible mechanism to define data exchange and to create intercommunications between the parameters of different simulation modules is in place. The solution fulfills the requirements set in the beginning of the work. Due to time and effort constraints, the approach used in the development was to reuse what is already present in the CluSys platform. For that reason, the parameter *label* and *module* name system is used to compare the parameter to the database for recognition. This approach however could pose problems in some cases. As the simulation modules are developed in different simulation environments, they have different *labeling* systems. In some cases, the module labels are generated automatically at runtime in CluSys, which could be a problem while defining the parameters in the database. A suggestion for this future problem is to implement a *reference label* system or assigning a unique *ID* to each simulation module when it is integrated in the CluSys platform. This *ID* will remain same throughout the lifetime of the simulation module and could be used instead of module name.

The interface can perform data adaptations or *transformations* before defining the intercommunication between the parameters. Three types of data *transformations* are implemented in the interface, these are, scalar parameter direct transformation, characteristic curve direct transformation and characteristic curve combination transformation. This could be extended in future to more data adaptation techniques, for example characteristic curve *transpose* or a *data matrix transformation*.

The intercommunication database is the backbone of the interface logic, that's why it is important that the data stored in it is consistent and correct. There are validity checks for the entered data implemented in the application prior to saving it to the database. These checks are general and check only if the entries are not empty or duplicate. This however, still allows the users to delete the data or accidently enter garbage values. All the *Expert level* users are authorized to access the database. For development and implementation phase, it was important to allow access so that the database could be tested but for later on, the access could be restricted only to certain *Experts*, who would check the accuracy of entered data before making it available for the automatic interface.

The intercommunication database contains information about the parameters in the simulation models, this information could be used further to implement useful functionalities in CluSys. For example, there is a functionality in CluSys, which allows

users to compare the values assigned to the parameters of two similar modules in one project. The comparison is done using the labels of the parameters. This comparison allows the user to compare the results in accordance with the used values. The intercommunication database, however, could allow the user to perform an “intelligent compare” between the parameters of different simulation modules. This comparison would be based on their physical meaning rather than their similar name in two modules and therefore would be independent of their module names.

Bibliography

[1] G. Rill and T. Schaeffer, Grundlagen und Methodik der Mehrkörpersimulation. 2nd Edition, Vieweg: Springer, 2014.

[2] O. Simonova, Entwicklung einer Software-Umgebung zur Auslegung einer 2VR-Doppelkupplung (Diploma Thesis), Hochschule Karlsruhe, Technik und Wirtschaft, 2010.

[3] R. D. Smith, "Simulation Article," 1998. [Online]. Available: http://www.modelbenders.com/encyclopedia/encyclopedia.html. [Accessed 31 January 2017].

[4] J. Banks, Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice, John Wiley & Sons, 1998.

[5] "LAC K1 Assembly," 28 November 2016. [Online]. Available: [Intranet resource].

[6] A. Albers and D. Herbst, "Chatter - Causes and Solutions," in *6th LuK Symposium 1998*, Bühl, Germany, 1998.

[7] "KES - Kinematics of Engagement System," Simulation Development Wiki, [Online]. Available: [Intranet resource]. [Accessed 28 November 2016].

[8] "TUCAN - Torque Fluctuation Analysis," Simulation Development Wiki, [Online]. Available: [Intranet resource]. [Accessed 28 11 2016].

[9] "SimScape Multibody: Model and simulate multibody mechanical systems," SimScape Multibody. MathWorks, Inc., [Online]. Available: https://de.mathworks.com/products/simmechanics.html. [Accessed 22 December 2016].

[10] "SIMPACK: Multibody Simulation Software," Simulia SIMPACK. Dassualt Systemes GmbH, [Online]. Available: http://www.simpack.com/ . [Accessed 22 December 2016].

[11] "SimulationX – Software for vehicle development," ESI ITI GmbH, [Online]. Available: https://www.simulationx.com/industries/simulation-automotive.html. [Accessed 26 January 2017].

[12] "MBDyn - Free MultiBody Dynamics Simulation Software," [Online]. Available: https://www.mbdyn.org/ . [Accessed 20 December 2016].

[13] "MODEL.Connect: Model.CONNECT - AVL's Co-Simulation Platform," MODEL.Connect. AVL North America, [Online]. Available: https://www.avl.com/-/model-connect-. [Accessed 10 January 2017].

[14] "EuroSim: Overview," EuroSim, [Online]. Available: http://www.eurosim.nl/. [Accessed 22 December 2016].

[15] "Parameter Exchange Editor: Reference Manual," EuroSim, [Online]. Available: http://www.eurosim.nl/support/manuals/manual_4_3/html/SUM/8_Parameter_Exchange.html#ch:pxe. [Accessed 22 December 2016].

[16] J. L. Harrington, Relational Database Design and Implementation, 3rd Edition, Morgan Kaufmann, 2009.

[17] K. Klöckner, Im Vergleich: NoSQL vs. relationale Datenbanken, Siegen: Universität Siegen, 2015.

[18] E. R. and S. B. Navathe, Fundamentals of Database Systems, 4th Edition, Boston: Addison Wesley, 2004.

[19] P. I. T. Wohayo, Entwicklung eines Frameworks zur Visualisierung von Simulationsergebnissen der Engineering-Werkzeuge (Bachelor Thesis), Hochschule Pforzheim, 2015.

[20] K. Kerr, "C++: The Most Powerful Language for .NET Framework Programming," Microsoft Developers Network, Microsoft Corporation, July 2004. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms379617(v=vs.80).aspx [Accessed 12 December 2016].

[21] "Overview of .NET Framework," Microsoft Developers Network , Microsoft Corporation, [Online]. Available: https://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx. [Accessed 12 December 2016].

[22] ".NET Framework Class Library," Microsoft Developers Network, Mocrosoft Coorporation, [Online]. Available: https://msdn.microsoft.com/de-de/library/gg145045(v=vs.110).aspx. [Accessed September 2016].

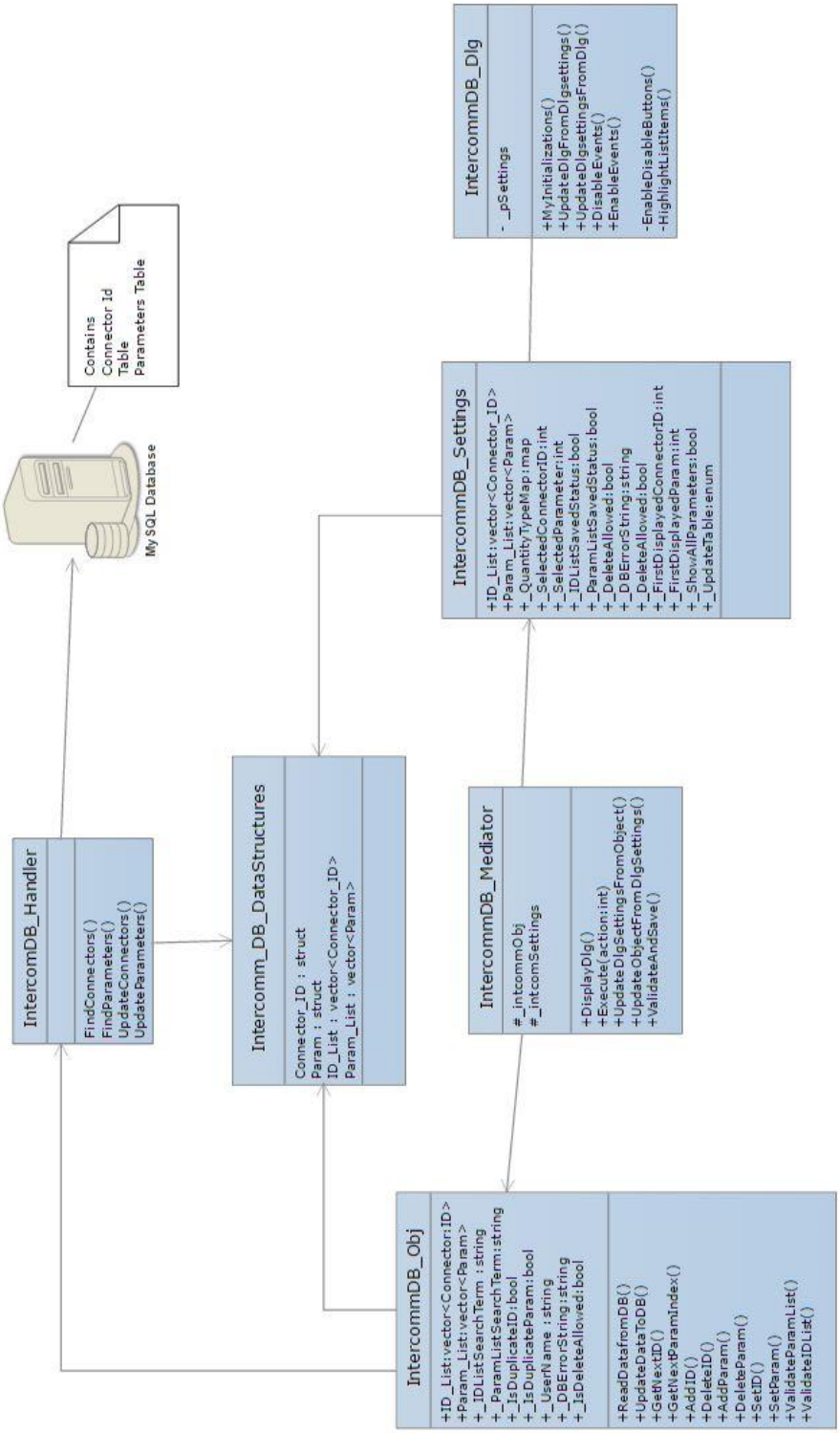
[23] B. Stroustrup, The Design and Evolution of C++, Pearson Education, 1994.

[24] "Lambda Expressions in C++," Microsoft Developers Network , Microsoft Corporation, [Online]. Available: https://msdn.microsoft.com/en-us/library/dd293608.aspx. [Accessed 12 December 2016].

[25] D. Kieras, "Using C++ Lambdas (Tutorial)," EECS Department, University of Michigan, 2015.

Appendix A

Class-Diagram of Intercommunication Database Application



IntercomDB_Obj methods:

AddID()

This function adds a Connector ID with a unique *_ID* value. This value is set based on the highest number of index in the list. A more complex approach could be used in the future if an integer value for *_ID* is not sufficient.

AddParam()

This function adds a new parameter to the Parameter List. The index of the parameter is also unique and based on the highest value of index in the list.

DeleteID()

This method receives an integer value as argument and deletes an ID from the list if the index is equal to the passed argument. Deletion is only possible if the Connector Id is not yet used in any parameter as a foreign key, or in other words is not yet used to define a parameter. In which case, an error message is received from the database.

DeleteParam()

It deletes the parameter if the passed argument equals an index in the list. This function also takes multiple selection in the form of an array of indices.

SetID()

This function checks if a Connector ID being edited in the dialog is valid or not. It checks if the name entered for the ID is unique. If not then it sets the value of _IsDuplicateID to true, so that the user could be informed.

SetParam()

This function checks if a parameter being edited in the dialog is valid or not. It checks if the combination of label and the module name entered for the parameter is unique. If not then it sets the value of _IsDuplicateParam to true, so that the user could be informed.

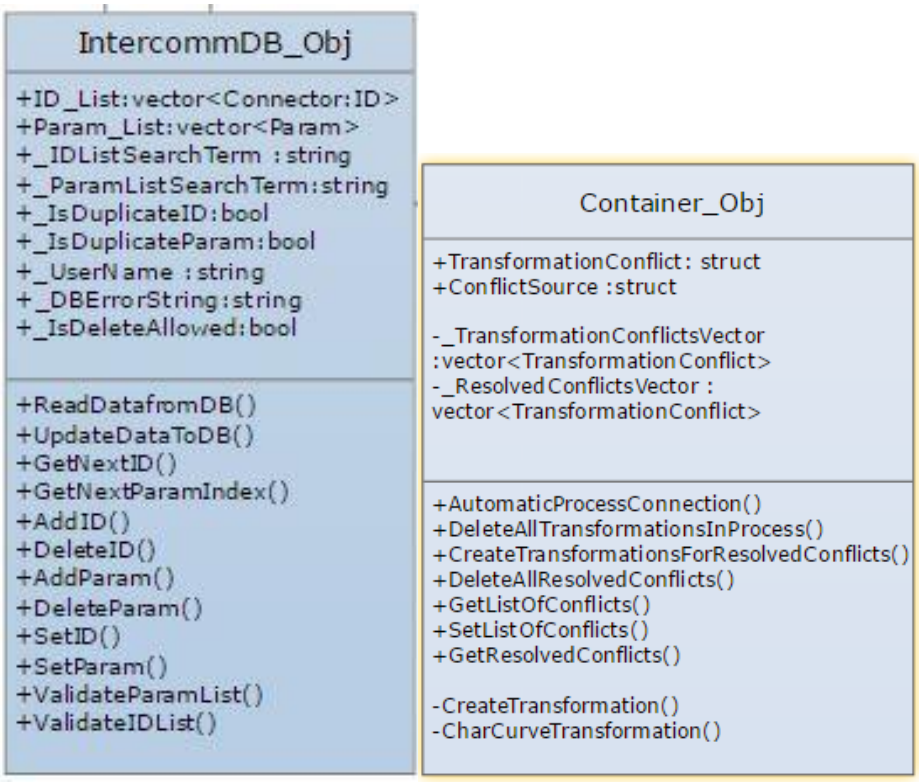
ValidateIDList()

This function validates the ID list by checking if there is any empty entry created in the table. If there is no empty entry, it checks if all the entries are unique. If there is any empty or duplicate entry in the ID list, the function returns a Boolean false.

ValidateParamList()

This function validates the entries in the parameter list. It checks if there is any empty entry, if there is no empty entry, it checks if all the entries are unique. The *uniqueness*

is checked by comparing combination of parameter label and module name. If all the entries are unique, then the function returns a true, else false.



Appendix B

XML structure for saving *Resolved Conflicts:*

```
// save conflicts resolution for automatic intercommunication process
XMLElement xml_resolved_conflicts;
XMLElement xml_conflict;
XMLElement xml_possible_inputs;
XMLElement xml_input;

e.CreateChildElement("All_Conflicts", &xml_resolved_conflicts);


for (int i = 0; i < (int)_ResolvedConflictsVector.size(); i++)
{
	xml_resolved_conflicts.CreateChildElement("Conflict", &xml_conflict);

	xml_conflict.SetAttribute("ConflictType",
_ResolvedConflictsVector.at(i)._ConflictType);
	xml_conflict.SetAttribute("DestinationModuleLabel",
_ResolvedConflictsVector.at(i)._DestinationModuleLabel);
	xml_conflict.SetAttribute("Index", _ResolvedConflictsVector.at(i)._Index);
	xml_conflict.SetAttribute("IsResolved", _ResolvedConflictsVector.at(i)._IsResolved);
	xml_conflict.SetAttribute("outputLabel",
_ResolvedConflictsVector.at(i)._outputLabel);
	xml_conflict.SetAttribute("ReferenceLabel",
_ResolvedConflictsVector.at(i)._ReferenceLabel);
	xml_conflict.SetAttribute("OutputDenomination",
_ResolvedConflictsVector.at(i)._OutputDenomination);
	xml_conflict.SetAttribute("ResolvedConflictStatus",
	_ResolvedConflictsVector.at(i)._ResolvedConflictStatus);
```

```
        xml_conflict.CreateChildElement("PossibleInputs", &xml_possible_inputs);

for (int j = 0; j < (int)_ResolvedConflictsVector.at(i)._PossibleInputs.size(); j++)
        {
        xml_possible_inputs.CreateChildElement("Input", &xml_input);

        xml_input.SetAttribute("Denomination",
_ResolvedConflictsVector.at(i)._PossibleInputs.at(j)._Denomination);
        xml_input.SetAttribute("IsSelected",
_ResolvedConflictsVector.at(i)._PossibleInputs.at(j)._IsSelected);
        xml_input.SetAttribute("ReferenceLabel",
_ResolvedConflictsVector.at(i)._PossibleInputs.at(j)._ReferenceLabel);
        xml_input.SetAttribute("Label",
_ResolvedConflictsVector.at(i)._PossibleInputs.at(j)._Label);
        }
}
```

XML structure for loading *Resolved Conflicts* from project data:

```
// load conflicts resolution for container process
_ResolvedConflictsVector.clear();
XMLElement          xml_resolved_conflicts;
XMLElement          xml_conflict;
XMLElementList      xml_conflict_list;
XMLElementList      xml_possible_inputs;

if(e.GetXMLElementByPath(&e,"All_Conflicts", xml_resolved_conflicts))
{
        xml_resolved_conflicts.GetElementList(&xml_conflict_list);
}

XMLElementList::iterator itr = xml_conflict_list.begin();
for( ; itr != xml_conflict_list.end(); itr++ )
{
        XMLElement *element = *itr;
```

```
	TransformationConflict conflict;

	conflict._ConflictType = element->GetAttributeAsInteger("ConflictType");
	conflict._DestinationModuleLabel = element-
>GetAttributeAsString("DestinationModuleLabel");
	conflict._Index= element->GetAttributeAsInteger("Index");
	conflict._IsResolved   = element->GetAttributeAsBool("IsResolved");
	conflict._outputLabel = element->GetAttributeAsString("outputLabel");
	conflict._ReferenceLabel= element->GetAttributeAsString("ReferenceLabel");
	conflict._OutputDenomination= element-
>GetAttributeAsString("OutputDenomination");
	//conflict._ResolvedConflictStatus    = element-
>GetAttributeAsInteger("ResolvedConflictStatus");

if (element->GetXMLElementByPath(element, "PossibleInputs", xml_conflict))
{
	xml_conflict.GetElementList(&xml_possible_inputs);
}
XMLElementList::iterator itr2 = xml_possible_inputs.begin();
for( ; itr2 != xml_possible_inputs.end(); itr2++ )
{
	XMLElement *element2 = *itr2;

ConflictSource confsource;
confsource._Denomination    = element2->GetAttributeAsString("Denomination");
confsource._IsSelected= element2->GetAttributeAsBool("IsSelected");
confsource._ReferenceLabel = element2->GetAttributeAsString("ReferenceLabel");
confsource._Label      = element2->GetAttributeAsString("Label");
conflict._PossibleInputs.push_back(confsource);
}
xml_possible_inputs.clear();
_ResolvedConflictsVector.push_back(conflict);
}
```